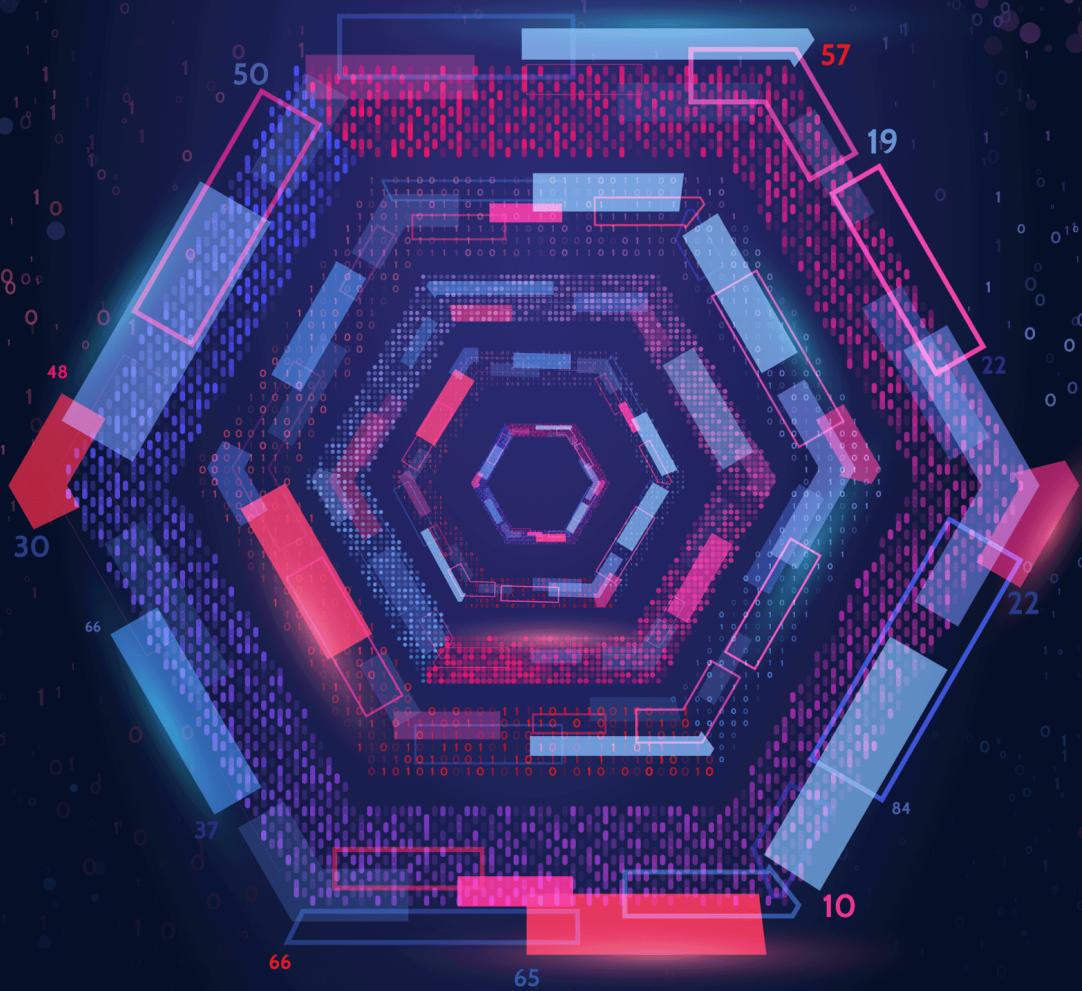


# Занурення в **ПАТЕРНИ ПРОЕКТУВАННЯ**



Олександр Швець

# **Занурення в ПАТЕРНИ ПРОЕКТУВАННЯ**

v2021-2.40

Книга придбана Сергій Бойчура  
[boychura95@gmail.com](mailto:boychura95@gmail.com) (#102537)

# Замість копірайту

Привіт! Мене звуть Олександр Швець, я автор книги Занурення в Патерни, а також онлайн-курсу Занурення в Рефакторинг.



Ця книга призначена для вашого особистого користування. Будь ласка, не передавайте її третім особам, за винятком членів своєї сім'ї. Якщо ви хочете поділитися книгою з друзями чи колегами – придбайте і подаруйте їм легальну копію книги. Також ви можете придбати корпоративну ліцензію для всієї вашої команди або організації.

Всі гроші, отримані з продажу моїх книг і курсів, ідуть на розвиток Refactoring.Guru. Це один з небагатьох ресурсів програмістської тематики, доступних українською мовою. Кожна придбана копія продовжує життя проекту й наближає момент виходу нового курсу чи книги.

© Олександр Швець, Refactoring.Guru, 2021

✉ [support@refactoring.guru](mailto:support@refactoring.guru)

🖼 Ілюстрації: Дмитро Жарт

🇺🇦 Переклад українською: Віталій Гальцев, Олександр Швець

📝 Редактор: Ельвіра Мамонтова

*Присвячую цю книгу своїй дружині Марії,  
без якої я б не довів діло до кінця ще  
років тридцять.*

# Зміст

<b>Зміст.....</b>	<b>4</b>
<b>Як читати цю книгу .....</b>	<b>6</b>
<b>ВСТУП ДО ООП.....</b>	<b>7</b>
Згадуємо ООП .....	8
Наріжні камені ООП .....	12
Зв'язки між об'єктами .....	19
<b>ОСНОВИ ПАТЕРНІВ .....</b>	<b>25</b>
Що таке патерн?.....	26
Навіщо знати патерни? .....	30
<b>ПРИНЦИПИ ПРОЕКТУВАННЯ.....</b>	<b>31</b>
Якості хорошої архітектури.....	32
<b>Базові принципи проектування .....</b>	<b>36</b>
§ Інкапсулюйте те, що змінюється .....	37
§ Програмуйте на рівні інтерфейсу.....	41
§ Віддавайте перевагу композиції перед спадкуванням .....	46
<b>Принципи SOLID .....</b>	<b>50</b>
§ S: Принцип єдиного обов'язку.....	51
§ O: Принцип відкритості/закритості.....	53
§ L: Принцип підстановки Лісков .....	56
§ I: Принцип поділу інтерфейсу.....	62
§ D: Принцип інверсії залежностей.....	65

<b>КАТАЛОГ ПАТЕРНІВ.....</b>	<b>68</b>
<b>Породжуvalльні патерни .....</b>	<b>69</b>
§ Фабричний метод / <i>Factory Method</i> .....	71
§ Абстрактна фабрика / <i>Abstract Factory</i> .....	87
§ Будівельник / <i>Builder</i> .....	101
§ Прототип / <i>Prototype</i> .....	118
§ Одинак / <i>Singleton</i> .....	132
<b>Структурні патерни.....</b>	<b>141</b>
§ Адаптер / <i>Adapter</i> .....	144
§ Міст / <i>Bridge</i> .....	157
§ Компонувальник / <i>Composite</i> .....	171
§ Декоратор / <i>Decorator</i> .....	184
§ Фасад / <i>Facade</i> .....	202
§ Легковаговик / <i>Flyweight</i> .....	212
§ Замісник / <i>Proxy</i> .....	226
<b>Поведінкові патерни .....</b>	<b>238</b>
§ Ланцюжок обов'язків / <i>Chain of Responsibility</i> .....	242
§ Команда / <i>Command</i> .....	260
§ Ітератор / <i>Iterator</i> .....	280
§ Посередник / <i>Mediator</i> .....	295
§ Знімок / <i>Memento</i> .....	310
§ Спостерігач / <i>Observer</i> .....	325
§ Стан / <i>State</i> .....	340
§ Стратегія / <i>Strategy</i> .....	356
§ Шаблонний метод / <i>Template Method</i> .....	369
§ Відвідувач / <i>Visitor</i> .....	381
<b>Заключення .....</b>	<b>395</b>

## Як читати цю книгу?

Ця книга складається з опису 22-х класичних патернів проектування, вперше відкритих «Бандою Чотирьох» (“Gang of Four” або просто GoF) у 1994 році.

Кожен розділ книги присвячений тільки одному патерну. Саме тому книгу можна читати як послідовно, від краю до краю, так і в довільному порядку, вибираючи тільки ті патерни, які вас цікавлять на даний момент.

Більшість патернів пов’язані між собою, тому ви зможете з легкістю стрибати по пов’язаних темах, використовуючи величезну кількість гіперпосилань, якими всіяні всі розділи книги. В кінці кожного розділу наведені відносини поточного патерна з іншими. Якщо ви бачите там назву патерна, до якого ще не дійшли, продовжуйте читати далі, цей пункт буде повторено в іншому розділі.

Патерни проектування універсальні. Тому всі приклади коду у цій книзі наведено на псевдокоді, без прив’язки до конкретної мови програмування.

Перед вивченням патернів ви можете освіжити пам’ять, проїшовшись основними термінами об’єктного програмування. Паралельно я розповім про UML-діаграми, яких у цій книзі приведено вдосталь. Якщо ви все це вже знаєте, сміливо приступайте до вивчення патернів.

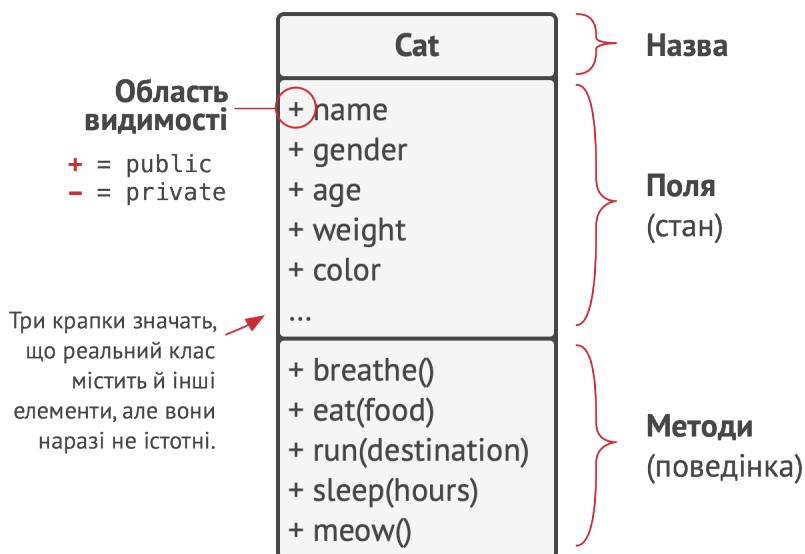
# **ВСТУП ДО ООП**

# Згадуємо ООП

**Об'єктно-орієнтоване програмування** – це методологія програмування, в якій усі важливі речі представлені **об'єктами**, кожен з яких є екземпляром того чи іншого **класу**, а класи утворюють **ієрархію** успадкування.

## Об'єкти, класи

Ви любите кошенят? Сподіваюсь, що любите, тому я спробую пояснити усі ці речі на прикладах з котами.



Це UML-діаграма класу. У книзі буде багато таких діаграм.

Отже, у вас є кіт Пухнастик. Він є *об'єктом класу Кіт*. Усі коти мають одинаковий набір властивостей: ім'я, стать, вік, вагу, колір, улюблену їжу та інше. Це – *поля* класу.

Крім того, всі коти поводяться схожим чином: бігають, дихають, сплять, їдять і муркочуть. Все це – *методи* класу. Узагальнено, поля і методи іноді називають *членами* класу.

Значення полів певного об'єкта зазвичай називають *його станом*, а сукупність методів – *поведінкою*.



**Pushystyk: Cat**

```
name    = "Pushystyk"
sex     = "male"
age     = 3
weight  = 5.5
color   = gray
```



**Murka: Cat**

```
name    = "Murka"
sex     = "female"
age     = 1
weight  = 3.5
color   = white
```

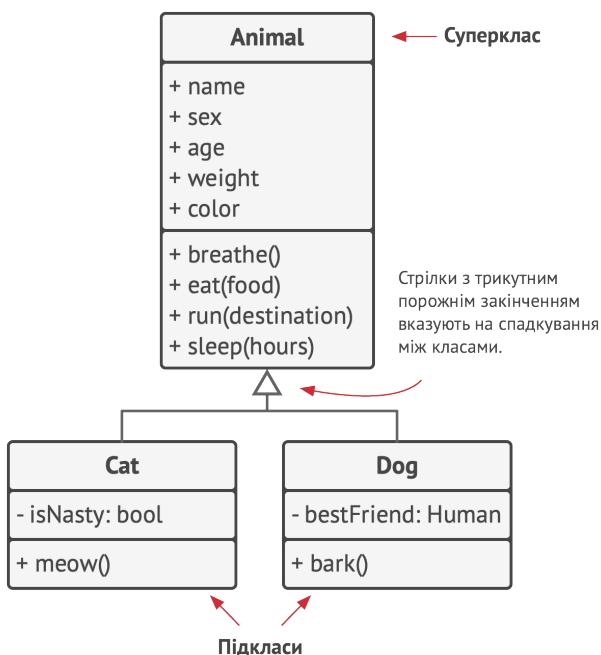
*Об'єкти – це екземпляри класів.*

Мурка, кішка вашої подруги, теж є екземпляром класу *Кіт*. Вона має такі самі властивості та поведінку, що й Пухнастик,

а відрізняється від нього лише значеннями цих властивостей – вона іншої статі, має інший колір, вагу тощо. Отже, **клас** – це своєрідне «креслення», на підставі якого будуються **об'єкти** – екземпляри цього класу.

## Ієрархії класів

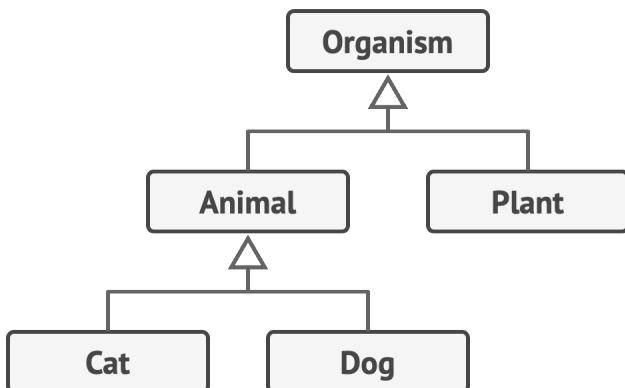
Ідемо далі. У вашого сусіда є собака Жучка. Як відомо, і собаки, і коти мають багато спільного: ім'я, стать, вік, колір є не тільки в котів, але й у собак. Крім того, бігати, дихати, спати та їсти можуть не тільки коти. Виходить так, що ці властивості та поведінка притаманні усьому класу **Тварини**.



UML-діаграма ієрархії класів. Усі класи на цій діаграмі є частиною ієрархії **Тварин**.

Такий батьківський клас прийнято називати **супер класом**, а його нащадків – **під класами**. Під класи успадковують властивості й поведінку свого батька, тому в них міститься лише те, чого немає у супер класі. Наприклад, тільки коти можуть муркотіти, а собаки – гавкати.

Ми можемо піти далі та виділити ще більш загальний клас живих **Організмів**, який буде батьківським і для **Тварин**, і для **Риб**. Таку «піраміду» класів зазвичай називають **ієрархією**. Клас **Котів** успадкує все, як з **Тварин**, так і з **Організмів**.



*Класи на UML-діаграмі можна спрощувати, якщо важливіше показати зв'язки між ними.*

Варто згадати, що під класи можуть перевизначати поведінку методів, які їм дісталися від супер класів. При цьому вони можуть, як повністю замінити поведінку методу, так і просто додати щось до результату виконання батьківського методу.

# Наріжні камені ООП

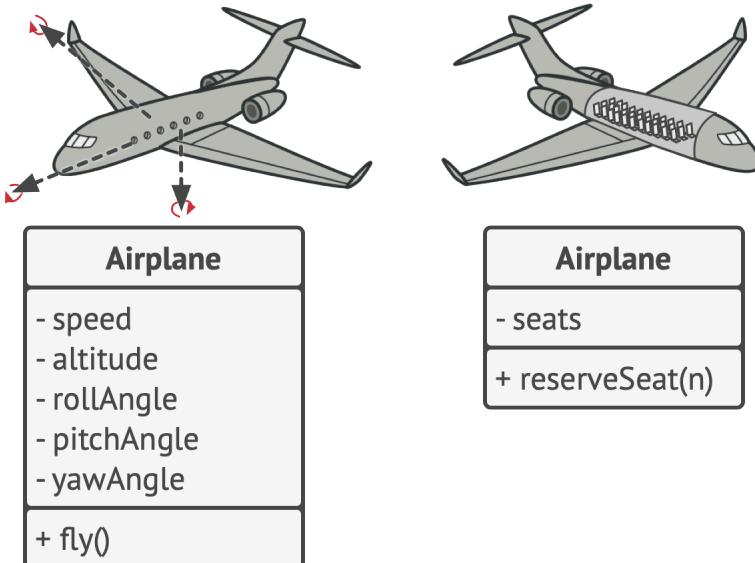
ООП має чотири головні концепції, які відрізняють його від інших методологій програмування.



## Абстракція

Коли ви пишете програму, використовуючи ООП, ви подаєте її частини через об'єкти реального світу. Але об'єкти у програмі не повторюють точно своїх реальних аналогів, та це й не завжди потрібно. Замість цього об'єкти програми всього лише *моделюють* властивості й поведінку реальних об'єктів, важливих у конкретному контексті, а інші – ігнорують.

Так, наприклад, клас `Літак` буде актуальним, як для програми-тренажера пілотів, так і для програми бронювання авіаквитків, але в першому випадку будуть важливими деталі пілотування літака, а в другому – лише розташування та наявність вільних місць усередині літака.



Різні моделі одного й того самого реального об'єкта.

**Абстракція** – це модель деякого об'єкта або явища реально-го світу, яка відкидає незначні деталі, що не грають істотної ролі в даному контексті.

## Інкапсуляція

Коли ви заводите автомобіль, достатньо повернути ключ запалювання або натиснути відповідну кнопку. Вам не потрі-бно вручну з'єднувати дроти під капотом, повернати колінча-стий вал та поршні, запускаючи такт двигуна. Всі ці деталі приховані під капотом автомобіля. Вам доступний лише простий інтерфейс: ключ запалювання, кермо та педалі. Таким чином, ми отримуємо визначення *інтерфейсу* – публі-чної (`public`) частини об'єкта, що доступна іншим об'єктам.

Інкапсуляція – це здатність об'єктів приховувати частину свого стану й поведінки від інших об'єктів, надаючи зовнішньому світові тільки визначений інтерфейс взаємодії з собою.

Наприклад, ви можете *інкапсулювати* щось всередині класу, зробивши його *приватним* (`private`) та приховавши доступ до цього поля чи методу для об'єктів інших класів. Трохи більш вільний, *захищений* (`protected`) режим видимості зробить це поле чи метод доступним у підкласах.

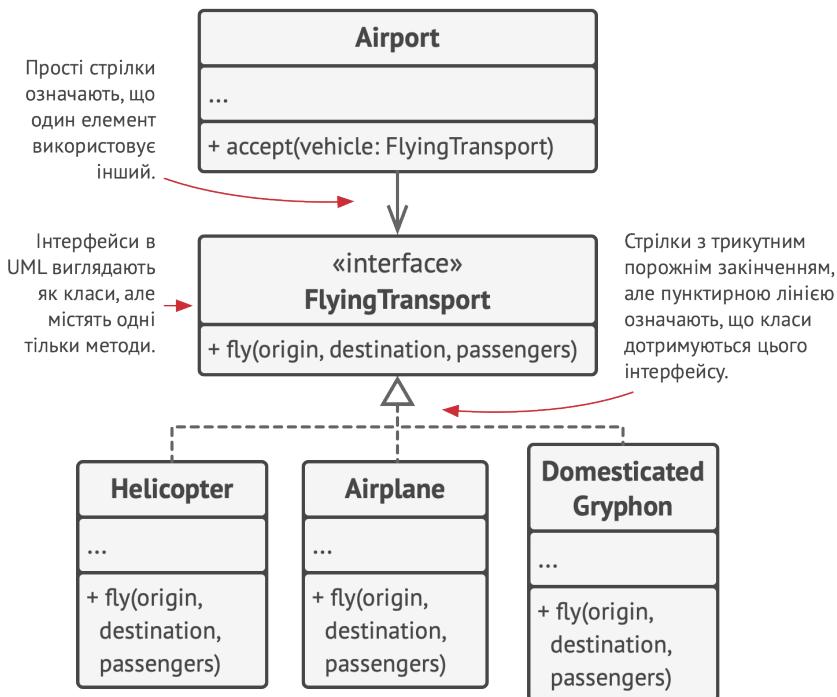
На ідеях абстракції та інкапсуляції побудовано механізми інтерфейсів і абстрактних класів/методів більшості об'єктних мов програмування.

Багатьох вводить в оману те, що словом «інтерфейс» називають і публічну частину об'єкта, і конструкцію `interface` більшості мов програмування.

В об'єктних мовах програмування за допомогою механізму інтерфейсів, які зазвичай оголошують через ключове слово `interface`, можна явно описувати «контракти» взаємодії об'єктів.

Наприклад, ви створили інтерфейс `ЛітаючийТранспорт` з методом `летіти(звідки, куди, пасажири)`, а потім описали методи класу `Аеропорт` так, щоб вони приймали будь-які об'єкти з цим інтерфейсом. Тепер ви можете бути впевнені

в тому, що будь-який об'єкт, який реалізує інтерфейс чи то **Літак**, **Вертоліт** чи **ДресированийГрифон**, зможе працювати з **Аеропортом**.



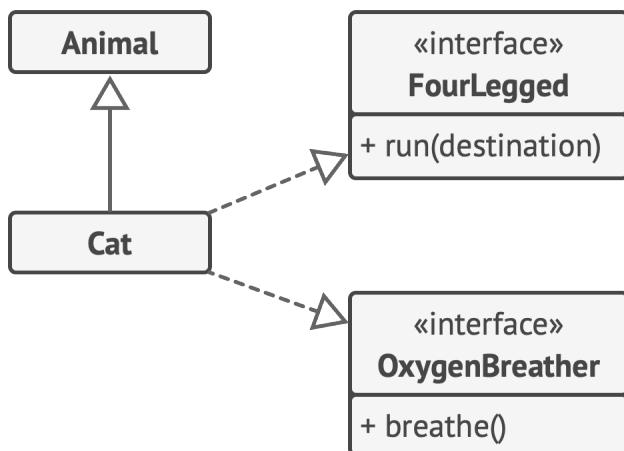
*UML-діаграма реалізації та використання інтерфейсу.*

Ви можете як завгодно змінювати код класів, що реалізують інтерфейс, не турбуючись про те, що **Аеропорт** втратить сумісність з ними.

## Спадкування

**Спадкування** – це можливість створення нових класів на основі існуючих. Головна користь від спадкування – повто-

рне використання існуючого коду. Розплата за спадкування виражається в тому, що підкласи завжди дотримуються інтерфейсу батьківського класу. Ви не можете виключити з підкласу метод, оголошений його предком.

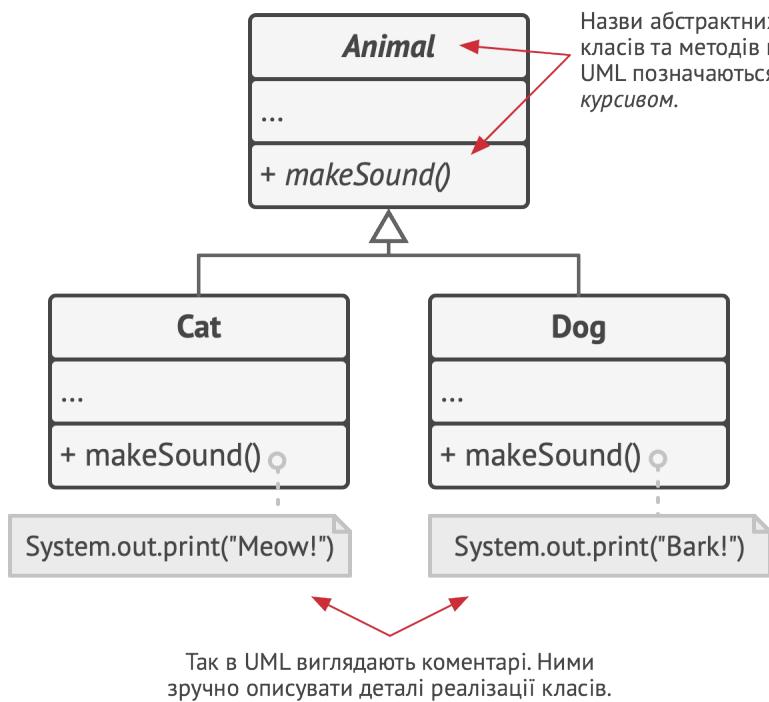


*UML-діаграма одиничного спадкування проти реалізації безлічі інтерфейсів.*

У більшості об'єктних мов програмування підклас може мати тільки одного «батька». Але, з іншого боку, клас може реалізовувати декілька інтерфейсів одночасно.

## Поліморфізм

Повернемося до прикладів з тваринами. Практично всі **Тварини** вміють видавати звуки, тому ми можемо оголосити їхній спільний метод видавання звуків *абстрактним*. Усі підкласи повинні будуть перевизначити та реалізувати такий метод по-своєму.



Тепер уявіть, що ми спочатку помістили декількох собак і котів у здоровезний мішок, а потім із закритими очима будемо витягувати їх з мішка одне за одним. Витягнувши тваринку, ми не знаємо достеменно її класу. Але, якщо її

погладити, тваринка видасть звук залежно від її конкретного класу.

```
1 bag = [new Cat(), new Dog()];
2
3 foreach (Animal a : bag)
4     a.makeSound()
5
6 // Meow!
7 // Bark!
```

Тут програмі невідомий конкретний клас об'єкта змінної `a`, але завдяки спеціальному механізму, що називається *поліморфізм*, буде запущено той метод видавання звуків, який відповідає реальному класу об'єкта.

*Поліморфізм* – це здатність програми вибирати різні реалізації під час виклику операцій з однією і тією ж назвою.

Для кращого розуміння *поліморфізм* можна розглядати як здатність об'єктів «прикидатися» чимось іншим. У вищезгаданому прикладі собаки й коти прикидалися абстрактними тваринами.

# Зв'язки між об'єктами

Окрім спадкування та реалізації існує ще декілька видів зв'язків між об'єктами, про які ми ще не говорили.

## Залежність



Залежність в UML-діаграмах. Професор залежить від навчального курсу.

Залежність це базовий зв'язок між класами, який показує, що один клас швидше за все доведеться міняти при зміні назви або сигнатури методів другого. Залежність з'являється там, де ви вказуєте конкретні назви класів – у викликах конструкторів, під час опису типів параметрів і значень методів тощо. Ступінь залежності можна послабити, якщо замість конкретних класів посилатися на абстрактні класи чи інтерфейси.

Зазвичай UML-діаграма не показує всі залежності – їх занадто багато в будь-якому реальному коді. Замість забруднення діаграми залежностями, ви повинні бути дуже прискіпливими і показувати лише ті залежності, що важливі для змісту, який ви хочете донести.

## Асоціація



*Асоціація в UML-діаграмах. Професор взаємодіє зі студентом.*

Асоціація – це коли один об'єкт взаємодіє з іншим. В UML асоціація позначається звичайною стрілкою, що спрямована в сторону взаємодії. Двостороння асоціація між об'єктами теж цілком прийнятна. Асоціацію можна розглядати як більш суворий варіант залежності, в якому один об'єкт завжди має доступ до об'єкта, з яким він взаємодіє. Водночас, під час простої залежності зв'язок може бути не постійним та не таким явним.

Наприклад, якщо один клас має поле-посилання на інший клас, ви можете відобразити цей зв'язок асоціацією. Цей зв'язок постійний, бо один об'єкт завжди може дотукатися до іншого через це поле. Причому, роль поля може відігравати і метод, який повертає об'єкти певного класу.

Щоб остаточно зрозуміти різницю між асоціацією та залежністю, давайте подивимося на комбінований приклад. Уявіть, що в нас є клас **Професор** :

```

1 class Professor is
2   field Student student
3   // ...
4   method teach(Course c) is
5     // ...
6     this.student.remember(c.getKnowledge())

```

Зверніть увагу на метод `navchiti`, що приймає аргумент класу `Курс`, який далі використовується в тілі методу. Якщо метод `отриматиЗнання` класу `Курс` змінить назву, чи в ньому з'являться якісь обов'язкові параметри, чи ще щось – наш код зламається. Це – залежність.

Тепер подивіться на поле `студент` та на те, як це поле використовується в методі `navchiti`. Ми можемо точно сказати, що клас `Студент` для професора також є залежністю, бо якщо метод `запам'ятати` змінить назву, то код професора теж зламається. Але завдяки тому, що значення поля `студент` доступне для професора завжди, з будь-якого методу, клас `Студент` – це не просто залежність, але ще й асоціація.

## Агрегація



Агрегація в UML-діаграмах. Кафедра містить професорів.

**Агрегація** – це спеціалізований різновид асоціації, що описує зв'язки *один-до-багатьох, багато-до-багатьох, частини-ціле* між декількома об'єктами.

Зазвичай під час агрегації один об'єкт *містить* інші, тобто виступає контейнером або колекцією. Тут контейнер не керує життєвим циклом компонентів і компоненти цілком можуть існувати окремо від контейнера.

В UML агрегація позначається лінією зі стрілкою на одному кінці та порожнім ромбом на іншому. Ромб спрямований в бік контейнера, а стрілка – в сторону компонента.

Пам'ятайте, що хоча ми говоримо про зв'язки між об'єктами, блоки на UML-діаграмі зображають зв'язки між класами. Об'єкт університету може складатися з декількох відділів, але ви побачите лише один блок відділу на діаграмі. UML дозволяє вказувати кількість об'єктів по обидві сторони зв'язків, але їх можна опустити, якщо кількість і так зрозуміла із контексту.

## Композиція



Композиція в UML-діаграмах. Університет складається з кафедр.

Композиція – це більш суворий варіант агрегації, коли один об'єкт складається з інших. Особливість цього зв'язку полягає в тому, що компонент може існувати лише як частина контейнера. В UML композиція зображується так само як і агрегація, але з зафарбованим ромбом.

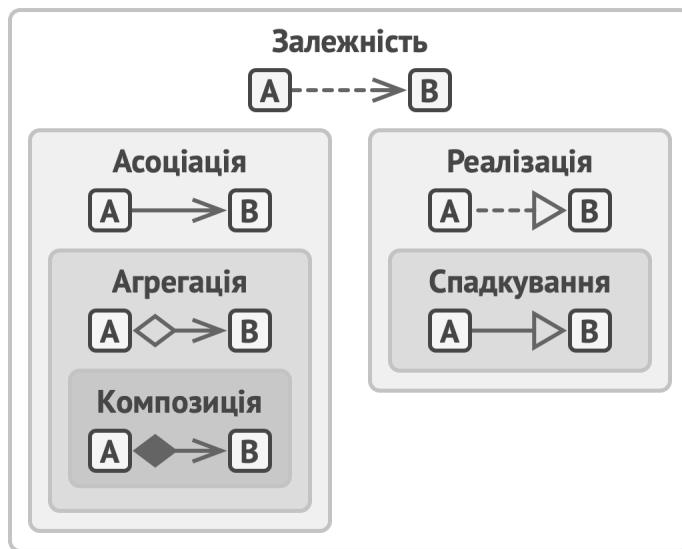
Зверніть увагу, у звичайному спілкуванні дуже часто під терміном «композиція» може матися на увазі як сама композиція, так і більш слабка агрегація. Справа в тому, що англійська фраза «object composition» означає буквально складений з об'єктів. Ось чому звичайну колекцію об'єктів часто-густо можуть називати *побудованою на принципах композиції*.

## Загальна картина

Тепер, коли ми знаємо про всі типи зв'язків, можна поглянути як вони пов'язані між собою. Це позбавить вас від плутанини та питань на кшталт «чим агрегація відрізняється від композиції» або «чи є спадкування залежністю».

- **Залежність:** Клас А можуть торкнутися зміни в класі В.
- **Асоціація:** Об'єкт А знає про об'єкт В. Клас А залежить від В.
- **Агрегація:** Об'єкт А знає про об'єкт В і складається з нього. Клас А залежить від В.
- **Композиція:** Об'єкт А знає про об'єкт В, складається з нього і керує його життєвим циклом. Клас А залежить від В.

- **Реалізація:** Клас А визначає методи оголошенні інтерфейсом В. Об'єкти А можна розглядати через інтерфейс В. Клас А залежить від В.
- **Спадкування:** Клас А успадковує інтерфейс та реалізацію класу В, але може перевизначити її. Об'єкти А можна розглядати через інтерфейс класу В. Клас А залежить від В.



Зв'язки між об'єктами та класами – від найслабших до найсильніших.

# **ОСНОВИ ПАТЕРНІВ**

# Що таке патерн?

**Патерн проектування** – це типовий спосіб вирішення певної проблеми, що часто зустрічається при проектуванні архітектури програм.

На відміну від готових функцій чи бібліотек, патерн не можна просто взяти й скопіювати в програму. Патерн являє собою не якийсь конкретний код, а загальний принцип вирішення певної проблеми, який майже завжди треба підлаштовувати для потреб тієї чи іншої програми.

Патерни часто плутають з алгоритмами, адже обидва поняття описують типові рішення відомих проблем. Але якщо алгоритм – це чіткий набір дій, то патерн – це високорівневий опис рішення, реалізація якого може відрізнятися у двох різних програмах.

Якщо провести аналогії, то алгоритм – це кулінарний рецепт з чіткими кроками, а патерн – інженерне креслення, на якому намальовано рішення без конкретних кроків його отримання.

## ☰ 3 чого складається патерн?

Описи патернів зазвичай дуже формальні й найчастіше складаються з таких пунктів:

- проблема, яку вирішує патерн;
- мотивація щодо вирішення проблеми способом, який пропонує патерн;
- структура класів, складових рішення;
- приклад однією з мов програмування;
- особливості реалізації в різних контекстах;
- зв'язки з іншими патернами.

Такий формалізм опису дозволив зібрати великий каталог патернів, додатково перевіривши кожен патерн на дієвість.

## Класифікація патернів

Патерни відрізняються за рівнем складності, деталізації та охоплення проектованої системи. Проводячи аналогію з будівництвом, ви можете підвищити безпеку на перехресті, встановивши світлофор, а можете замінити перехрестя цілою автомобільною розв'язкою з підземними переходами.

Найбільш низькорівневі та прості патерни – *ідіоми*. Вони не дуже універсальні, позаяк мають сенс лише в рамках однієї мови програмування.

Найбільш універсальні – *архітектурні патерни*, які можна реалізувати практично будь-якою мовою. Вони потрібні для проектування всієї програми, а не окремих її елементів.

Крім цього, патерни відрізняються і за призначенням. У цій книзі буде розглянуто три основні групи патернів:

- **Породжуєчі патерни** піклуються про гнучке створення об'єктів без внесення в програму залежностей.
- **Структурні патерни** показують різні способи побудови зв'язків між об'єктами.
- **Поведінкові патерни** піклуються про ефективну комунікацію між об'єктами.

## Хто вигадав патерни?

За визначенням, патерни не вигадують, а радше «відкривають». Це не якісь супер-оригінальні рішення, а, навпаки, типові способи вирішення однієї і тієї ж проблеми, що часто повторюються з невеликими варіаціями.

Концепцію патернів вперше описав Крістофер Александер у книзі *Мова шаблонів. Міста. Будівлі. Будівництво*<sup>1</sup>. У книзі описано «мову» для проектування навколишнього середовища, одиниці якого – шаблони (або *патерни*, що близче до оригінального терміна *patterns*) – відповідають на архітектурні запитання: якої висоти потрібно зробити вікна, скільки поверхів має бути в будівлі, яку площу в мікрорайоні відвести для дерев та газонів.

---

1. A Pattern Language: Towns, Buildings, Construction:  
<https://refactoring.guru/uk/pattern-language-book>

Ідея видалася привабливою четвірці авторів: Еріху Гаммі, Річарду Хелму, Ральфу Джонсону, Джону Вліссідесу. У 1994 році вони написали книгу *Патерни проектування: повторно використовувані елементи архітектури об'єктно-орієнтованого програмного забезпечення*<sup>1</sup>, до якої увійшли 23 патерни, що вирішують різні проблеми об'єктно-орієнтованого дизайну. Назва книги була занадто довгою, щоб хтось зміг її запам'ятати. Тому незабаром усі стали називати її “book by the gang of four”, тобто «книга від банди чотирьох», а потім і зовсім “GoF book”.

З того часу було знайдено десятки інших об'єктних патернів. «Патерновий» підхід став популярним і в інших галузях програмування, тому зараз можна зустріти різноманітні патерни також за межами об'єктного проектування.

- 
1. *Design Patterns: Elements of Reusable Object-Oriented Software:*  
<https://refactoring.guru/uk/gof-book>

# Навіщо знати патерни?

Ви можете цілком успішно працювати, не знаючи жодного патерна. Більше того, ви могли вже не раз реалізувати який-небудь з патернів, навіть не підозрюючи про це.

Але якраз свідоме володіння інструментом відрізняє професіонала від аматора. Ви можете забити цвях молотком, а можете й дрилем, якщо дуже сильно постараєтесь. Але професіонал знає, що головна фішка дриля зовсім не в цьому. Отже, навіщо ж знати патерни?

- **Перевірені рішення.** Ви витрачаєте менше часу, використовуючи готові рішення, замість повторного винаходу велосипеда. До деяких рішень ви могли б дійти й самотужки, але багато які з них стануть для вас відкриттям.
- **Стандартизація коду.** Ви робите менше прорахунків при проектуванні, використовуючи типові уніфіковані рішення, оскільки всі приховані в них проблеми вже давно знайдено.
- **Загальний словник програмістів.** Ви вимовляєте назву патерна, замість того, щоб годину пояснювати іншим програмістам, який крутий дизайн ви придумали і які класи для цього потрібні.

# **ПРИНЦИПИ ПРОЕКТУВАННЯ**

# Якості хорошої архітектури

Перш ніж перейти до вивчення конкретних патернів, поговорімо про сам процес проектування, про те, до чого треба прагнути і чого потрібно уникати.

## Повторне використання коду

Не секрет, що вартість і час розробки – це найбільш важливі метрики при розробці будь-яких програмних продуктів. Чим менші обидва ці показники, тим більш конкурентним продукт буде на ринку і тим більше прибутку отримає розробник.

**Повторне використання** програмної архітектури та коду – це один з найбільш поширених способів зниження вартості розробки. Логіка проста: замість того, щоб розробляти щось повторно, чому б не використати минулі напрацювання у новому проекті?

Ідея виглядає чудово на папері, але, на жаль, не весь код можна пристосувати до роботи в нових умовах. Занадто тісні зв'язки між компонентами, залежність коду від конкретних класів, а не абстрактних інтерфейсів, вшиті в код операції, які неможливо розширити, – все це зменшує гнучкість вашої архітектури та перешкоджає її повторному використанню.

На допомогу приходять патерни проектування, які ціною ускладнення коду програми підвищують гнучкість її частин, що полегшує подальше повторне використання коду.

Наведу цитату Еріха Гамми<sup>1</sup>, одного з першовідкривачів патернів, про повторне використання коду та ролі патернів у ньому.

“

Існує три рівні повторного використання коду. На самому нижньому рівні знаходяться класи: корисні бібліотеки класів, контейнери, а також «команди» класів типу контейнерів/ітераторів.

Фреймворки стоять на найвищому рівні. В них важливою є тільки архітектура. Вони визначають ключові абстракції для вирішення деяких бізнес-завдань, представлені у вигляді класів і відносин між ними. Візьміть JUnit, це дуже маленький фреймворк. Він містить усього декілька пов'язаних між собою класів: `Test`, `TestCase` та `TestSuite`. Зазвичай фреймворк має набагато більший обсяг, ніж один клас. Ви вклинуєтесь у фреймворк, розширяючи декотрі його класи. Все працює за так званим голлівудським принципом: «не телефонуйте нам, ми самі вам зателефонуємо». Фреймворк дозволяє вам задати якусь свою поведінку, а потім, коли приходить черга щось робити, сам викликає її. Те ж саме відбувається і в JUnit. Він звертається до вашого класу, коли

- 
1. Erich Gamma on Flexibility and Reuse: <https://refactoring.guru/gamma-interview>

потрібно виконати тест, але все інше відбувається всередині фреймворка.

Є ще середній рівень. Це те, де я бачу патерни. Патерни проектування менші за об'ємом та більш абстрактні, ніж фреймворки. Вони, насправді, є просто описом того, як парочка класів відноситься і взаємодіє один з одним. Рівень повторного використання підвищується, коли ви рухаєтесь в напрямку від конкретних класів до патернів, а потім до фреймворків.

Ще одною привабливою рисою цього середнього рівня є те, що патерни – це менш ризикований спосіб повторного використання, ніж фреймворки. Розробка фреймворку – це вкрай ризикована й дорога інвестиція. У той же час патерни дозволяють повторно використовувати ідеї та концепції у відриві від конкретного коду.

”

## Розширюваність

Зміни часто називають головним ворогом програміста.

- Ви придумали ідеальну архітектуру інтернет-магазину, але через місяць довелося додати інтерфейс для замовлень телефоном.
- Ви випустили відеогру під Windows, але потім знадобилася підтримка macOS.
- Ви зробили інтерфейсний фреймворк з квадратними кнопками, але клієнти почали просити круглі.

У кожного програміста кільканадцять подібних історій. Є кілька причин, чому так відбувається.

По-перше, всі ми починаємо розуміти проблему краще в процесі її вирішення. Нерідко до кінця роботи над першою версією програми ми вже готові повністю її переписати, оскільки стали краще розуміти деякі аспекти, які не були настільки нам зрозумілими спочатку. Зробивши другу версію, ви починаєте розуміти проблему ще краще, вносите ще зміни і так далі – процес не зупиняється ніколи, адже не тільки ваше розуміння, але ще й та сама проблема може змінитися з часом.

По-друге, зміни можуть прийти ззовні. У вас є ідеальний клієнт, який з первого разу сформулював те, що йому потрібно, а ви все це зробили. Чудово! Аж ось виходить нова версія операційної системи, в якій ваша програма перестає працювати. Бідкаючись, ви лізете в код, щоб внести деякі зміни.

Проте, на це все можна дивитися оптимістично: якщо хтось просить вас щось змінити в програмі, отже, вона комусь все ж таки ще потрібна.

Ось чому вже навіть трохи досвідчений програміст проектує архітектуру й пише код з урахуванням майбутніх змін.

# Базові принципи проектування

Що таке хороший дизайн? За якими критеріями його оцінювати, і яких правил дотримуватися при розробці? Як забезпечити достатній рівень гнучкості, зв'язаності, керованості, стабільності та зрозуміlostі коду?

Все це правильні запитання, але дляожної програми відповідь буде трохи відрізнятися. Давайте розглянемо універсальні принципи проектування, які допоможуть вам формулювати відповіді на ці запитання самостійно.

До речі, більшість патернів, наведених у цій книзі, базується саме на перерахованих нижче принципах.

## Інкапсулюйте те, що змінюється

Визначте аспекти програми, класу або методу, які змінюються найчастіше, і відокремте їх від того, що залишається постійним.

Цей принцип має на меті зменшити наслідки, викликані змінами. Уявіть, що ваша програма – це корабель, а зміни – то підступні міни на його шляху. Натикаючись на міну, корабель заповнюється водою та тоне.

Знаючи це, ви можете розділити трюм корабля на незалежні секції, проходи між якими наглуго зчиняти. Тепер після зіткнення з міною корабель залишиться на плаву. Вода затопить лише одну секцію, залишивши решту без змін.

Ізолюючи мінливі частини програми в окремих модулях, класах або методах, ви зменшуєте кількість коду, якого торкнуться наступні зміни. Отже, вам потрібно буде витратити менше зусиль на те, щоб привести програму до робочого стану, налагодити та протестувати код, що змінився. Де менше роботи, там менша вартість розробки. А там, де менша вартість, там і перевага перед конкурентами.

### Приклад інкапсуляції на рівні методу

Припустімо, що ви розробляєте інтернет-магазин. Десь всередині вашого коду знаходиться метод `getOrderTotal`, що

розраховує фінальну суму замовлення з урахуванням розміру податку.

Ми можемо припустити, що код обчислення податків, імовірно, буде часто змінюватися. По-перше, логіка нарахування податку залежить від країни, штату й навіть міста, в якому знаходиться покупець. До того ж, розмір податку не стаєй і може змінюватися з часом.

Через ці зміни вам доведеться постійно торкатися методу `getOrderTotal`, який, насправді, не особливо цікавиться *деталями* обчислення податків.

```

1 method getOrderTotal(order) is
2     total = 0
3     foreach item in order.lineItems
4         total += item.price * item.quantity
5
6     if (order.country == "US")
7         total += total * 0.07 // US sales tax
8     else if (order.country == "EU"):
9         total += total * 0.20 // European VAT
10
11    return total

```

*ДО: правила обчислення податків змішані з основним кодом методу.*

Ви можете перенести логіку обчислення податків в окремий метод, приховавши деталі від оригінального методу.

```

1  method getOrderTotal(order) is
2      total = 0
3      foreach item in order.lineItems
4          total += item.price * item.quantity
5
6      total += total * getTaxAmount(order.country)
7
8      return total
9
10 method getTaxAmount(country) is
11     if (country == "US")
12         return 0.07 // US sales tax
13     else if (country == "EU")
14         return 0.20 // European VAT
15     else
16         return 0

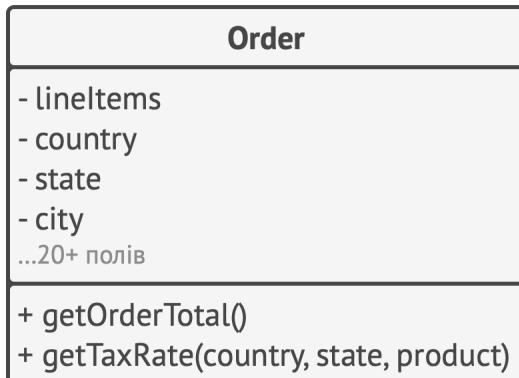
```

*ПІСЛЯ: розмір податку можна отримати, викликавши один метод.*

Тепер зміни податків будуть ізольовані в рамках одного методу. Більш того, якщо логіка обчислення податків ще більш ускладниться, вам буде легше отримати цей метод до власного класу.

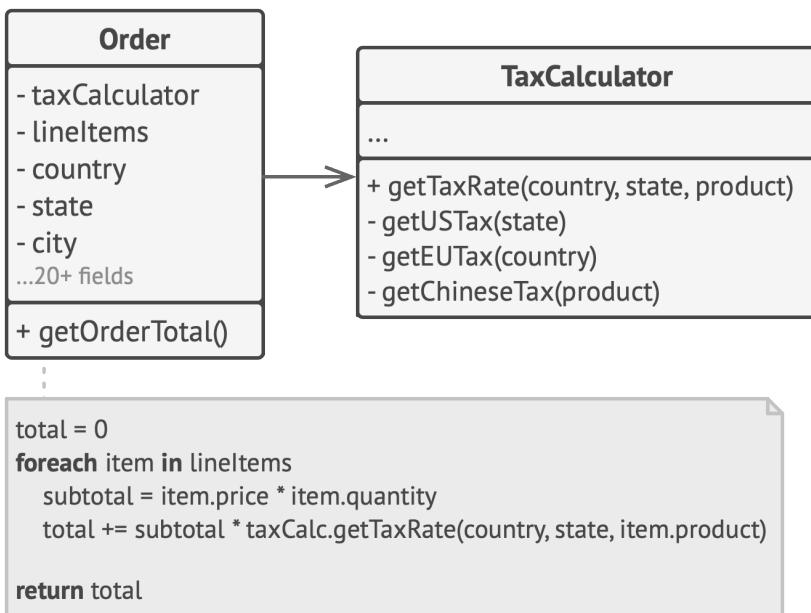
## Приклад інкапсуляції на рівні класу

Видобути логіку податків до власного класу? Якщо логіка податків стала занадто складною, то чому б і ні?



*ДО: обчислення податків у класі замовлень.*

Об'єкти замовлень делегуватимуть обчислення податків окремому об'єкту-калькулятору податків.



*ПІСЛЯ: обчислення податків приховано в класі замовлень.*

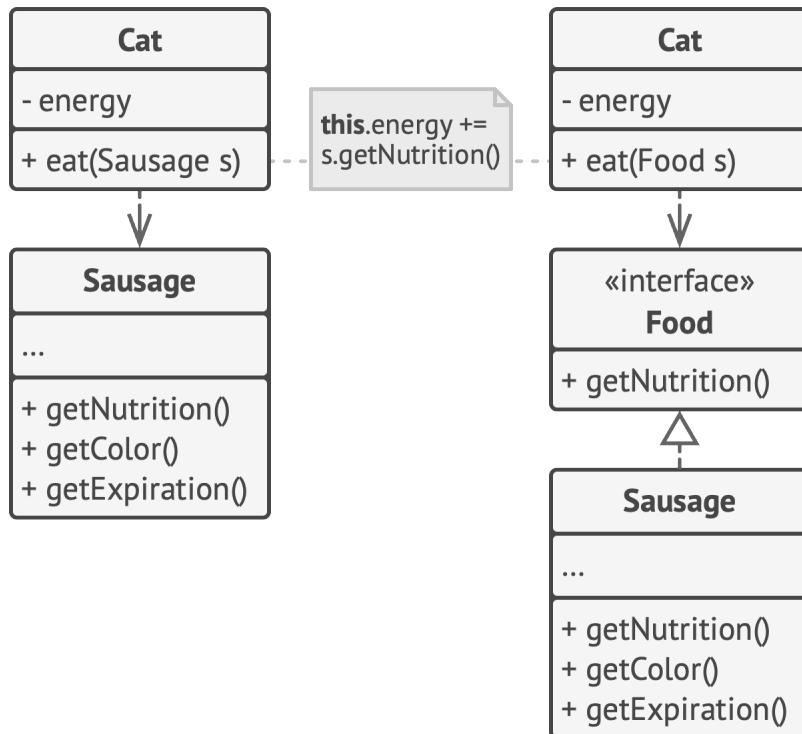
# Програмуйте на рівні інтерфейсу

Програмуйте на рівні інтерфейсу, а не на рівні реалізації. Код повинен залежати від абстракцій, а не від конкретних класів.

Гнучкість архітектури побудованої на класах виражається в тому, що їх можна легко розширювати, не ламаючи існуючий код. Для прикладу повернемося до класу котів. Клас `Кіт`, який єсть тільки сардельки, буде менш гнучким, ніж той, який може їсти будь-яку їжу. При цьому останнього можна буде годувати й сардельками теж, адже вони є їжею.

Коли вам потрібно налагодити взаємодію між двома об'єктами різних класів, то простіше всього зробити один клас прямо залежним від іншого. Що й казати, якщо, зазвичай, я й сам з цього починаю. Але є й інший, більш гнучкий спосіб.

1. Визначте, що саме потрібно одному об'єкту від іншого, які методи він викликає.
2. Потім опишіть ці методи в окремому інтерфейсі.
3. Зробіть так, щоб клас-залежність дотримувався цього інтерфейсу. Скоріше за все, потрібно буде лише додати цей інтерфейс до опису класу.
4. Тепер ви можете зробити інший клас залежним від інтерфейсу, а не конкретного класу.



*До та після вилучення інтерфейсу.*

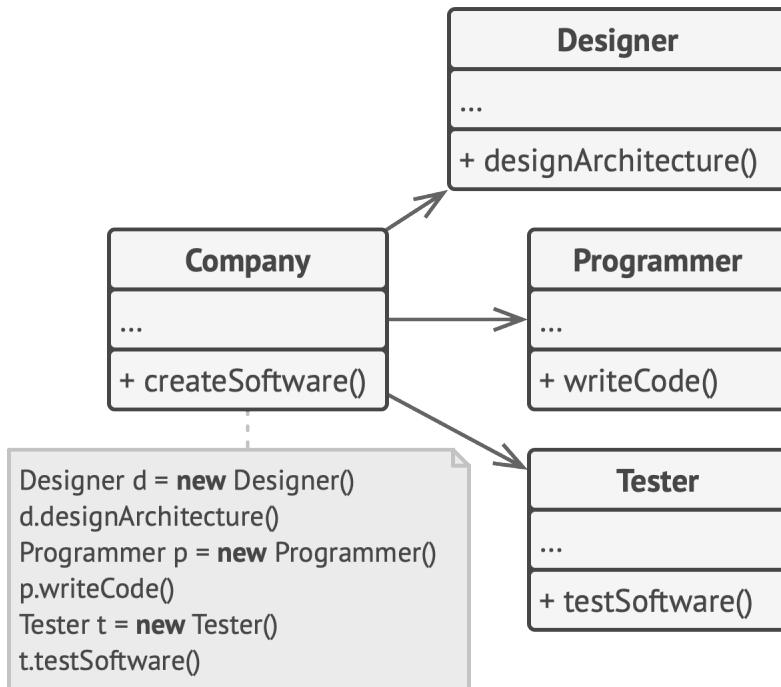
*Код праворуч більш гнучкий, але й більш складний від того коду, що ліворуч.*

Виконавши все це ви, імовірніше за все, не отримаєте миттєвої вигоди. Проте в майбутньому ви зможете використовувати альтернативні реалізації класів, не змінюючи код, що їх використовує.

## Приклад

Розгляньмо ще один приклад, де робота на рівні інтерфейсу виявляється кращою, ніж прив'язка до конкретних кла-

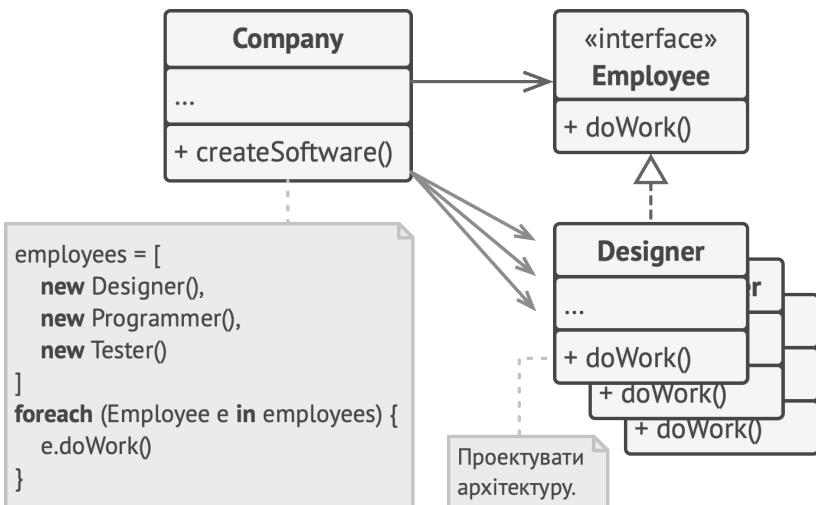
сів. Уявіть, що ви робите симулятор софтверної компанії. У вас є різні класи працівників, які виконують ту чи іншу роботу всередині компанії.



*ДО: класи жорстко пов'язані.*

Спочатку клас компанії жорстко прив'язаний до конкретних класів працівників. Попри те, що кожен тип працівників виконує різну роботу, ми можемо звести їхні методи роботи до одного виду, виділивши для всіх класів загальний інтерфейс.

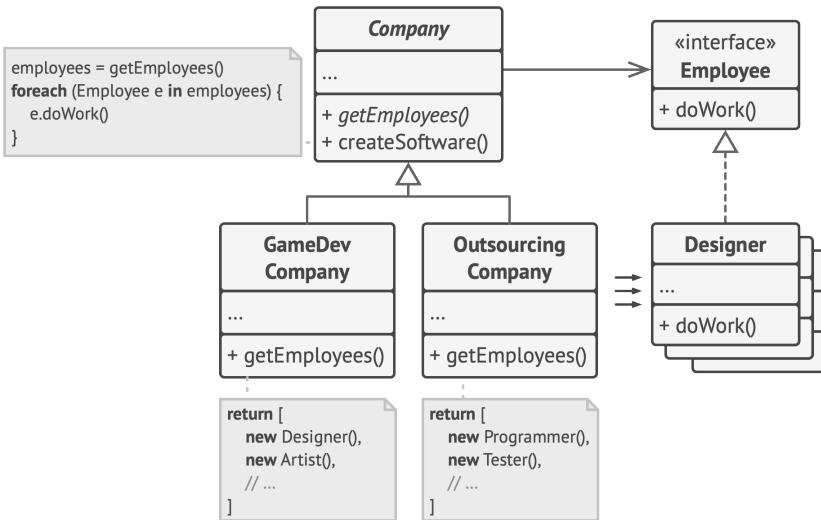
Зробивши це, ми зможемо застосувати поліморфізм у класі компанії, трактуючи всіх працівників однаково через інтерфейс `Employee`.



**КРАЩЕ:** поліморфізм допоміг спростити код, але основний код компанії все ще залежить від конкретних класів співробітників.

Тим не менше, клас компанії все ще залишається жорстко прив'язаним до конкретних класів працівників. Це не дуже добре, особливо, якщо припустити, що нам знадобиться реалізувати кілька видів компаній. Усі ці компанії відрізнятимуться конкретними працівниками, які їм потрібні.

Ми можемо зробити метод отримання працівників у базовому класі компанії *абстрактним*. Конкретні компанії повинні самі подбати про створення об'єктів співробітників. Отже, кожен тип компаній зможе мати власний набір співробітників.



**ПІСЛЯ:** основний код класу компанії став незалежним від класів співробітників. Конкретних співробітників створюють конкретні класи компаній.

Після цієї зміни код класу компанії став остаточно незалежним від конкретних класів. Тепер ми можемо додавати до програми нові види працівників і компаній, не вносячи зміни до основного коду базового класу компаній.

До речі, ви тільки що побачили приклад одного з патернів, а саме – *Фабричного методу*. Надалі ми ще повернемося до нього.

## Віддавайте перевагу композиції перед спадкуванням

Спадкування – це найпростіший та найшвидший спосіб повторного використання коду між класами. У вас є два класи з кодом, який дублюється. Створіть для них загальний базовий клас та перенесіть до нього спільну поведінку. Що може бути простішим?

Але у спадкування є і проблеми, які стають очевидними лише тоді, коли програма обросла класами, і змінити ситуацію вже досить важко. Ось деякі з можливих проблем зі спадкуванням.

- Підклас **не може відмовитися від інтерфейсу або реалізації** свого батька. Ви повинні будете реалізувати всі абстрактні методи батька, навіть якщо вони не потрібні для конкретного підкласу.
- Перевизначаючи методи батька, ви повинні **піклуватися про те, щоб не зламати базову поведінку** супер класу. Це важливо, адже підклас може бути використаний у будь-якому коді, що працює з супер класом.
- Спадкування **порушує інкапсуляцію супер класу**, оскільки підкласам доступні деталі батька. Супер класи можуть самі стати залежними від підкласів, наприклад, якщо програміст винесе до супер класу які-небудь загальні деталі підкласів, щоб полегшити подальше спадкування.

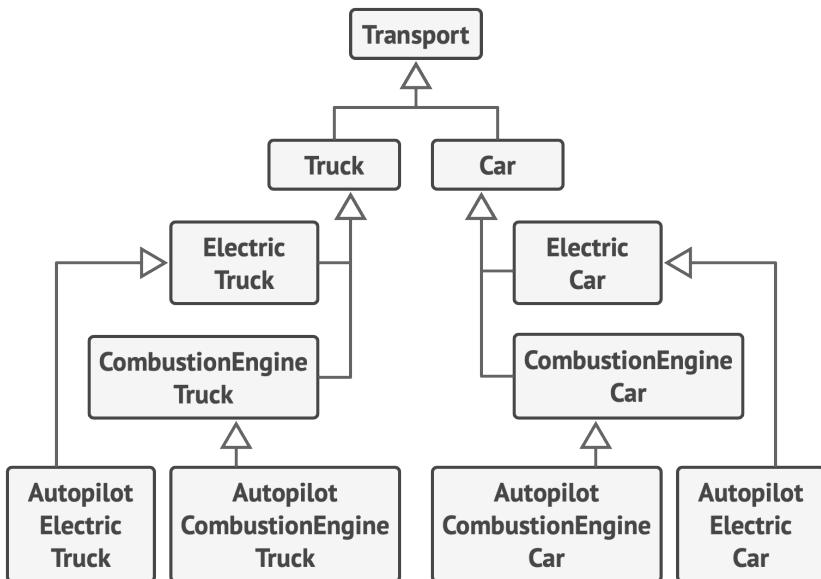
- Підкласи **дуже тісно пов'язані** з батьківським класом. Будь-яка зміна в батькові може зламати поведінку в підкласах.
- Повторне використання коду через наслідування може привести до **роздріблення ієархії класів**.

У наслідування є альтернатива, яка називається **композицією**. Якщо спадкування можна виразити словом «є» (автомобіль є транспортом), то композицію – словом «містить» (автомобіль *містить* двигун).

Цей принцип поширюється і на агрегацію – більш вільний вид композиції, коли два об'єкти є рівноправними, і жоден з них не керує життєвим циклом іншого. Оцініть різницю: автомобіль *містить* і водія, але той може вийти й пересісти до іншого автомобіля або взагалі піти пішки *самостійно*.

### Приклад

Припустімо, вам потрібно зmodелювати модельний ряд автовиробника. У вас є легкові автомобілі та вантажівки. Причому вони бувають з електричним двигуном та з двигуном на бензині. До того ж вони відрізняються режимами навігації – є моделі з ручним керуванням та автопілотом.

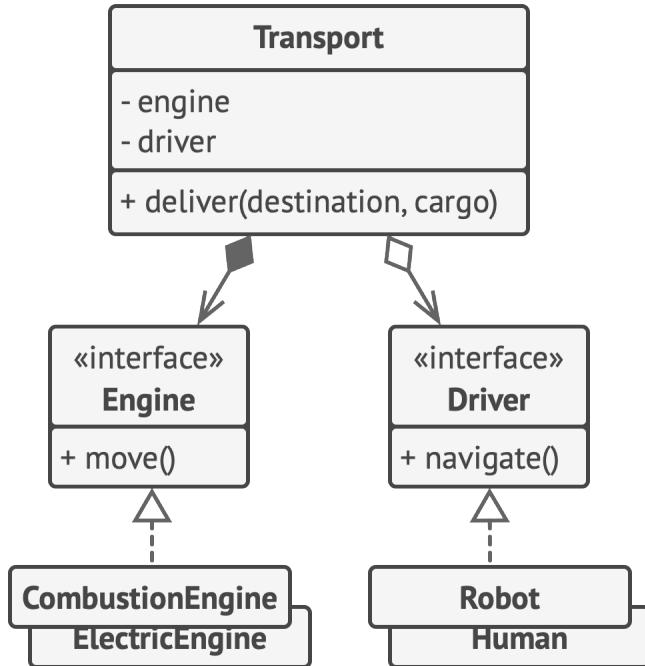


**СПАДКУВАННЯ:** розвиток класів у кількох площинах (*тип вантажу × тип двигуна × тип навігації*) призводить до комбінаторного вибуху.

Як бачите, кожен такий параметр призводить до збільшення кількості класів. Крім того, виникає проблема дублювання коду, тому що підкласи не можуть успадковувати декількох батьків одночасно.

Вирішити проблему можна за допомогою композиції. Замість того, щоб об'єкти самі реалізовували ту чи іншу поведінку, вони можуть делегувати її іншим об'єктам.

Композиція дає вам ще й іншу перевагу. Тепер, наприклад, ви можете замінити тип двигуна автомобіля безпосередньо під час виконання програми, підставивши в об'єкт транспорту інший об'єкт двигуна.



**КОМПОЗИЦІЯ:** різні види функціональності виділені у власні ієрархії класів.

Така структура властива патерну *Стратегія*, про який ми теж поговоримо у цій книзі.

# Принципи SOLID

Розглянемо ще п'ять принципів проектування, які відомі як SOLID. Вперше ці принципи були описані Робертом Мартіном у книзі *Agile Software Development, Principles, Patterns, and Practices*<sup>1</sup>.

Досягти такої лаконічності у назві вдалося шляхом використання невеличких хитрощів. Справа в тому, що термін SOLID – це абревіатура, за кожною буквою якої стоїть окремий принцип проектування.

Головна мета цих принципів – підвищити гнучкість вашої архітектури, зменшити пов'язаність між її компонентами та полегшити повторне використання коду.

Але, як і все в цьому житті, дотримання цих принципів має свою ціну. Тут це, зазвичай, виражається ускладненням коду програми. У реальному житті немає, мабуть, такого коду, в якому дотримувалися б усі ці принципи відразу. Тому пам'ятайте про баланс і не сприймайте все викладене як догму.

---

1. Agile Software Development, Principles, Patterns, and Practices:  
<https://refactoring.guru/uk/principles-book>

# S

## Принцип єдиного обов'язку Single Responsibility Principle

Клас має мати лише один мотив для зміни.

Намагайтесь досягти того, щоб кожен клас відповідав тільки за одну частину функціональності програми, причому вона повинна бути повністю інкапсульована в цей клас (читай, прихована всередині класу).

Принцип єдиного обов'язку призначений для боротьби зі складністю. Коли у вашій програмі всього 200 рядків, то дизайн, як такий, взагалі не потрібен. Достатньо охайно написати 5-7 методів, і все буде добре. Проблеми виникають тоді, коли система росте та збільшується в масштабах. Коли клас розростається, він просто перестає вміщуватися в голові. Навігація ускладнюється, на очі потрапляють непотрібні деталі, пов'язані з іншим аспектом, в результаті кількість понять починає перевищувати мозковий стек, і ви втрачаете контроль над кодом.

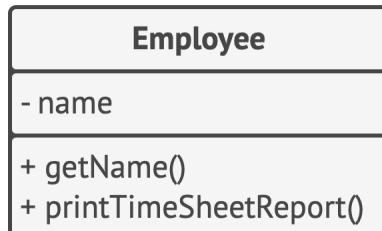
Якщо клас робить занадто багато речей одразу, вам доводиться змінювати його щоразу, коли одна з цих речей змінюється. При цьому є ризик пошкодження інших частин класу, яких ви навіть не планували торкатися.

Добре мати можливість зосередитися на складних аспектах системи окремо. Але, якщо вам складно це робити, застосо-

вуйте принцип єдиного обов'язку, розділяючи ваші класи на частини.

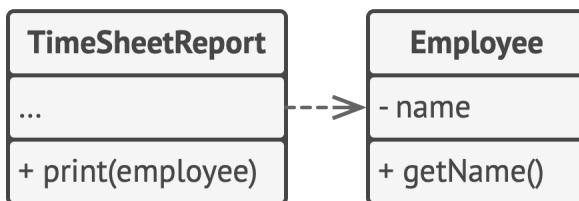
## Приклад

Клас `Employee` має відразу кілька причин для зміни. Перша пов'язана з головним завданням класу – керування даними співробітника. Але є й інша: зміни, пов'язані з форматуванням звіту для друку, зачіпатимуть клас співробітників.



*ДО: клас співробітника містить різномірні поведінки.*

Проблему можна вирішити, виділивши операцію друку в окремий клас.



*ПІСЛЯ: зайва поведінка переїхала до власного класу.*

# O Принцип відкритості/закритості pen/Closed Principle

Розширяйте класи, але не змінюйте їхній початковий код.

Прагніть досягти того, щоб класи були відкритими для розширення, але закритими для зміни. Головна ідея цього принципу в тому, щоб не ламати існуючий код при внесенні змін до програми.

Клас можна назвати відкритим, якщо він доступний для розширення. Наприклад, у вас є можливість розширити набір операцій або додати до нього нові поля, створивши власний підклас.

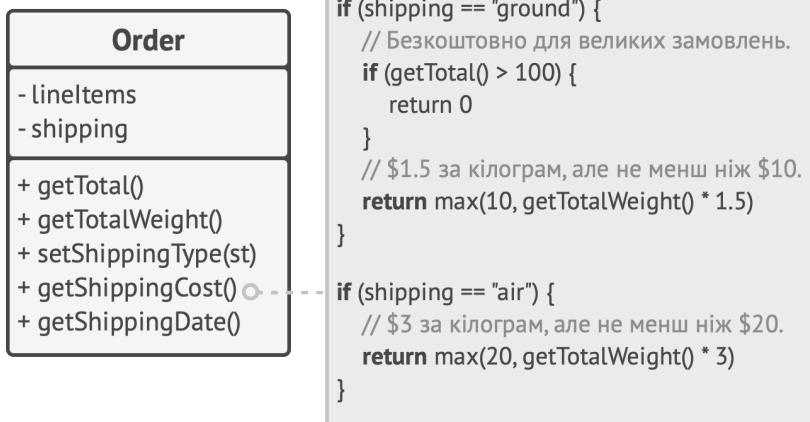
У той же час, клас можна назвати закритим (а краще скласти закінченим), якщо він готовий до використання іншими класами – його інтерфейс вже остаточно визначено, і він не змінюватиметься в майбутньому.

Якщо клас уже був написаний, схвалений, протестований, можливо, внесений до бібліотеки і включений до проекту, не бажано намагатися модифікувати його вміст після цього. Замість цього ви можете створити підклас і розширити в ньому базову поведінку, не змінюючи код батьківського класу безпосередньо.

Але не варто дотримуватись цього принципу буквально для кожної зміни. Якщо вам потрібно виправити помилку в початковому класі, просто візьміть і зробіть це. Немає сенсу вирішувати проблему батька в дочірньому класі.

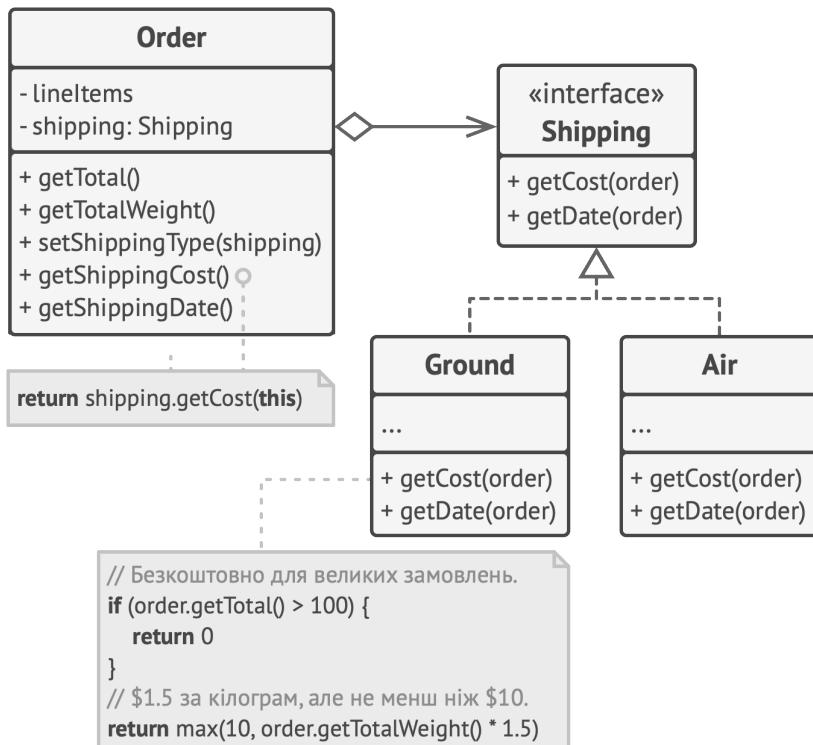
## Приклад

Клас замовлень має метод розрахунку вартості доставки, причому способи доставки «зашиті» безпосередньо в сам метод. Якщо вам потрібно буде додати новий спосіб доставки – доведеться зачіпати весь клас `Order`.



*ДО: код класу замовлення потрібно буде змінювати при додаванні нового способу доставки.*

Проблему можна вирішити, якщо застосувати патерн *Стратегія*. Для цього потрібно виділити способи доставки у власні класи з загальним інтерфейсом.



*ПІСЛЯ: нові способи доставки можна додати, не зачіпаючи клас замовлень.*

Тепер при додаванні нового способу доставки потрібно буде реалізувати новий клас інтерфейсу доставки, не зачіпаючи класу замовлень. Об'єкт способу доставки до класу замовлення буде подавати клієнтський код, який раніше встановлював спосіб доставки простим рядком.

Бонус цього рішення в тому, що розрахунок часу та дати доставки теж можна помістити до нових класів, підкоряючись принципу єдиного обов'язку.

# L Принцип підстановки Лісков liskov Substitution Principle<sup>1</sup>

Підкласи повинні доповнювати, а не підміняти поведінку базового класу.

Намагайтесь створювати підкласи таким чином, щоб їхні об'єкти можна було б підставляти замість базового класу, не ламаючи при цьому функціональність клієнтського коду.

Принцип підстановки – це ряд перевірок, які допомагають передбачити, чи залишиться підклас сумісним з іншим кодом програми, який успішно працював до цього, використовуючи об'єкти базового класу. Особливо це важливо під час розробки бібліотек та фреймворків, коли ваші класи використовуються іншими людьми, а ви не зможете впливати на чужий клієнтський код, навіть якщо б захотіли.

На відміну від інших принципів, які визначено дуже вільно, і вони мають безліч трактувань, принцип підстановки має певні формальні вимоги до підкласів, а точніше, до методів, перевизначеніх в них.

- 
1. Принцип названо на честь Барбари Лісков, котра вперше сформулювала його у 1987 році у роботі *Data abstraction and hierarchy*: <https://refactoring.guru/liskov/dah>

- **Типи параметрів методу підкласу повинні збігатися або бути більш абстрактними, ніж типи параметрів базового методу.** Звучить заплутано? Розглянемо, все на прикладі.
  - Базовий клас має метод `feed(Cat c)`, який вміє годувати хатніх котів. Клієнтський код це знає і завжди передає до методу кота.
  - **Добре:** Ви створили підклас і перевизначили метод годування так, щоб нагодувати будь-яку тварину: `feed(Animal c)`. Якщо підставити цей клас у клієнтський код — нічого поганого не станеться. Клієнтський код подасть до методу кота, але метод вміє годувати всіх тварин, тому нагодує і кота.
  - **Погано:** Ви створили інший підклас, в якому є метод, що вміє годувати виключно бенгальську породу котів (підклас котів): `feed(BengalCat c)`. Що буде з клієнтським кодом? Він так само подасть до методу звичайного кота, проте метод вміє годувати тільки бенгалів, тому не зможе відпрацювати, "зламавши" клієнтський код.
- **Тип значення методу підкласу, що повертається, повинен збігатися або бути підтиповим значенням базового методу, що повертається.** Тут все те саме, що і в попередньому пункті, але навпаки.
  - Базовий метод: `buyCat(): Cat`. Клієнтський код очікує на виході будь-якого хатнього кота.

- **Добре:** Метод підкласу: `buyCat(): BengalCat`. Клієнтський код отримає бенгальського кота, який є хатнім котом, тому все буде добре.
- **Погано:** Метод підкласу: `buyCat(): Animal`. Клієнтський код "зламається", оскільки незрозуміла тварина (можливо, крокодил) не поміститься у ящику для перенесення котів.

Ще один анти-приклад зі світу мов з динамічною типізацією: базовий метод повертає рядок, а перевизначений метод – число.

- **Метод не повинен викидати виключення, які не властиві базовому методу.** Типи виключень у перевизначеному методі повинні збігатися або бути підтипами виключень, які викидають базовий метод. Блоки `try-catch` у клієнтському коді спрямовані на конкретні типи виключень, що викидаються базовим методом. Тому несподіване виключення, викинуте підкласом, може проскочити скрізь обробника клієнтського коду та призвести до збою в програмі.

У більшості сучасних мов програмування, особливо строго типізованих (Java, C# та інші), переведовані обмеження вбудовано безпосередньо у компілятор. Тому при їхньому порушенні ви не зможете зібрати програму.

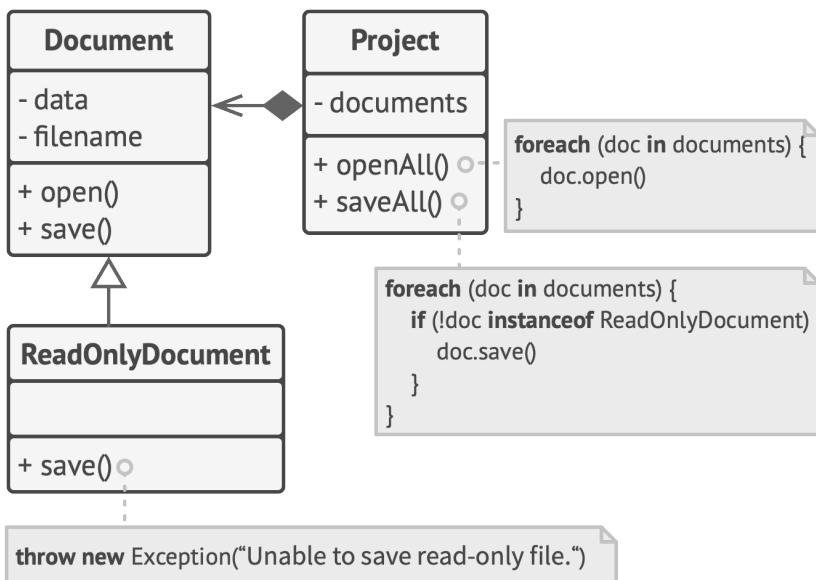
- **Метод не повинен посилювати перед-умови.** Наприклад, базовий метод працює з параметром типу `int`. Якщо підклас вимагає, щоб значення цього параметра було більшим за нуль, то це посилює вимоги передумови. Клієнтський код, який до цього відмінно працював, подаючи до методу негативні числа, тепер зламається при роботі з об'єктом підкласу.
- **Метод не повинен послаблювати пост-умови.** Наприклад, базовий метод вимагає, щоб після завершення методу всі підключення до бази даних було закрито, а підклас залишає ці підключення відкритими, щоб потім використовувати повторно. Проте клієнтський код базового класу нічого про це не знає. Він може завершити програму відразу після виклику методу, залишивши в системі запущені процеси-привиди.
- **Інваріанти класу повинні залишитися без змін.** Інваріант – це набір умов, за яких об'єкт має сенс. Наприклад, інваріант кота – це наявність чотирьох лап, хвоста, здатність муркотіти та інше. Інваріант може бути описано не тільки явно, контрактом або перевірками в методах класу, але й побічно, наприклад, юніт-тестами або клієнтським кодом.

Цей пункт легше за все порушити при спадкуванні, оскільки ви можете просто не підозрювати про існування якоїсь з умов інваріанта складного класу. Ідеальним був би підклас, який тільки вводить нові методи й поля, не торкаючись полів базового класу.

- Підклас не повинен змінювати значення приватних полів базового класу.** Цей пункт може звучати дивно, але в деяких мовах програмування доступ до приватних полів можна отримати через механізм рефлексії. В інших мовах, на кшталт Python та JavaScript, зовсім немає жорсткого захисту приватних полів.

## Приклад

Щоб закрити тему принципу підстановки, давайте розглянемо приклад невдалої ієархії класів документів.

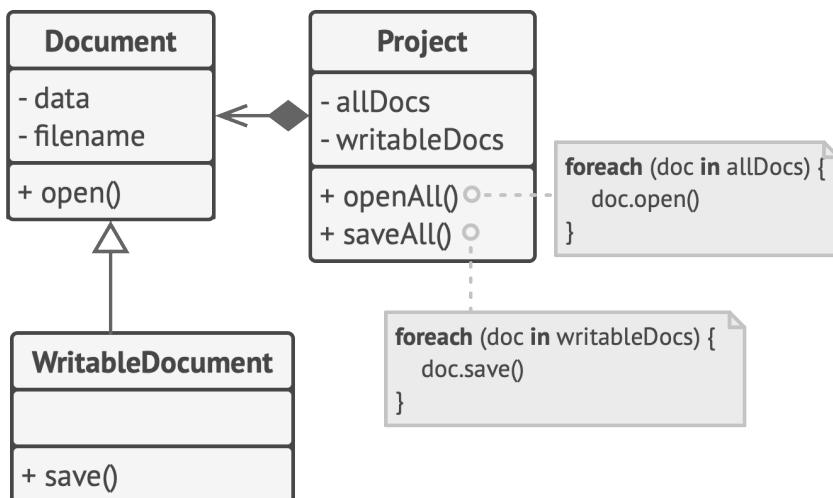


ДО: підклас «обнуляє» роботу базового методу.

Метод збереження в підкласі `ReadOnlyDocuments` викине виняток, якщо хтось намагатиметься викликати його метод

збереження. Базовий метод не має такого обмеження. Тому клієнтський код змушений перевіряти тип документа під час збереження всіх документів.

При цьому порушується ще й принцип відкритості/закритості, оскільки клієнтський код починає залежати від конкретного класу, який не можна замінити на інший, не вносячи змін до клієнтського коду.



*ПІСЛЯ:* підклас розширює базовий клас новою поведінкою.

Проблему можна вирішити, якщо перепроектувати ієархію класів. Базовий клас зможе тільки відкривати документи, але не матиме змоги зберігати їх. Підклас, який тепер називатиметься `WritableDocument`, розширить поведінку батькі-вського класу, дозволивши зберегти документ.

## I Принцип поділу інтерфейсу Interface Segregation Principle

Клієнти не повинні залежати від методів, які вони не використовують.

Прагніть досягти того, щоб інтерфейси були досить вузькими, а класам не доводилося б реалізовувати надмірну поведінку.

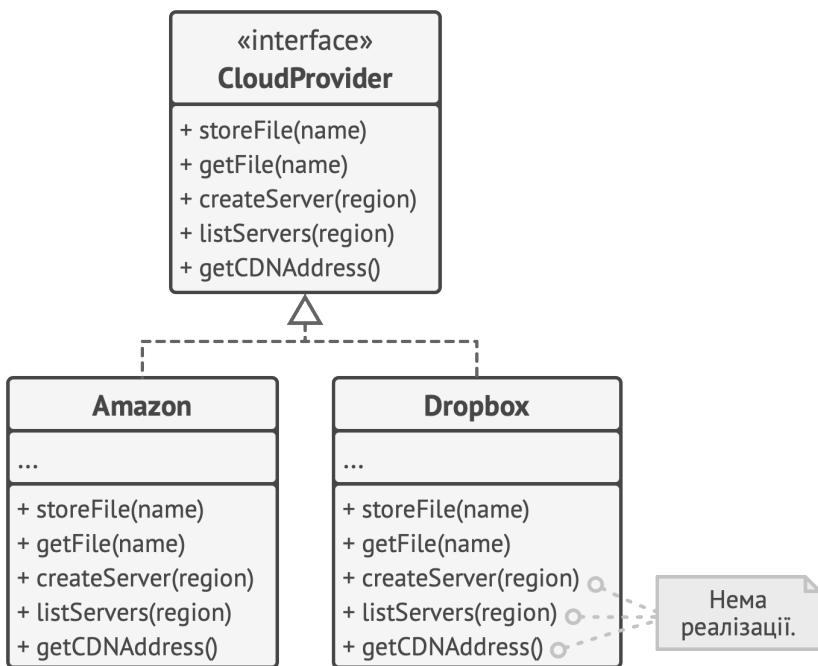
Принцип поділу інтерфейсів каже про те, що занадто «товсті» інтерфейси необхідно розділяти на більш маленькі й специфічні, щоб клієнти маленьких інтерфейсів знали тільки про методи, необхідні їм для роботи. В результаті при зміні методу інтерфейсу не повинні змінюватися клієнти, які цей метод не використовують.

Успадкування дозволяє класу мати тільки один супер клас, але не обмежує кількість інтерфейсів, які він може реалізувати. Більшість об'єктних мов програмування дозволяють класам реалізовувати відразу кілька інтерфейсів, тому немає потреби заштовхувати у ваш інтерфейс більше поведінок, ніж він того потребує. Ви завжди можете присвоїти класу відразу кілька менших інтерфейсів.

## Приклад

Уявіть бібліотеку для роботи з хмарним провайдерами. У першій версії вона підтримувала тільки Amazon, який має повний набір хмарних послуг. На підставі цього й проектувався інтерфейс майбутніх класів.

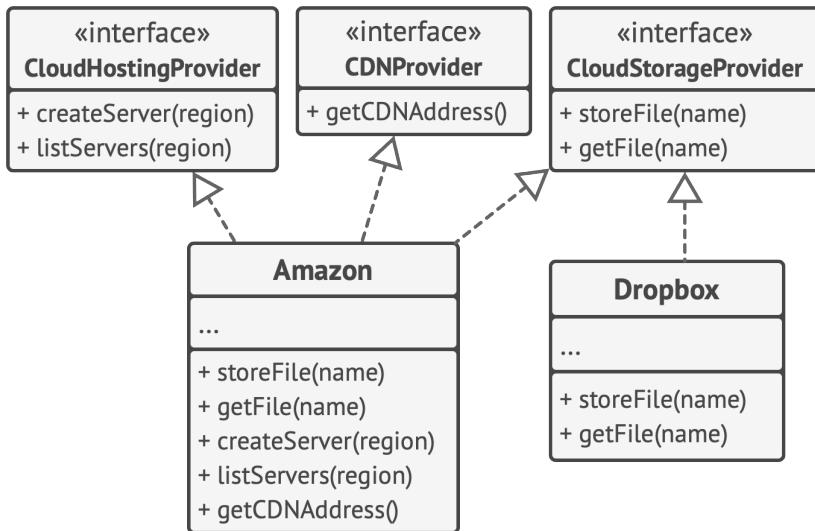
Але пізніше стало зрозуміло, що такий інтерфейс хмарного провайдера занадто широкий, оскільки є інші провайдери, які реалізують тільки частину з усіх доступних сервісів.



*ДО: не всі клієнти можуть реалізувати операції інтерфейсу.*

Щоб не плодити класи з порожньою реалізацією, роздутий інтерфейс можна розбити на частини. Класи, які були здатні

реалізувати всі операції старого інтерфейсу, можуть реалізувати відразу кілька нових часткових інтерфейсів.



*ПІСЛЯ: роздумтий інтерфейс розбитий на частини.*

**D**

## Принцип інверсії залежностей Dependency Inversion Principle

Класи верхніх рівнів не повинні залежати від класів нижніх рівнів. Обидва повинні залежати від абстракцій. Абстракції не повинні залежати від деталей. Деталі повинні залежати від абстракцій.

Зазвичай під час проектування програм можна виділити два рівні класів.

- **Класи нижнього рівня** реалізують базові операції на зразок роботи з диском, передачі даних мережею, підключення до бази даних та інше.
- **Класи високого рівня** містять складну бізнес-логіку програми, що спирається на класи низького рівня для здійснення більш простих операцій.

Здебільшого ви спочатку проектуєте класи низького рівня, а потім беретеся за верхній рівень. При такому підході класи бізнес-логіки стають залежними від більш примітивних низькорівневих класів. Кожна зміна в низькорівневому класі може зачепити класи бізнес-логіки, які його використовують.

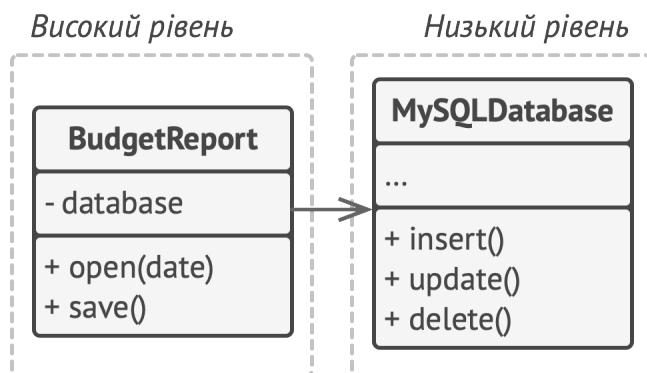
Принцип інверсії залежностей пропонує змінити напрямок, в якому відбувається проектування.

- Для початку вам потрібно описати інтерфейс низькорівневих операцій, які потрібні класу бізнес-логіки.
- Це дозволить вам прибрати залежність класу бізнес-логіки від конкретного низькорівневого класу, замінивши її «м'якою» залежністю від інтерфейсу.
- Низькорівневий клас, у свою чергу, стане залежним від інтерфейсу, визначеного бізнес-логікою.

Принцип інверсії залежностей часто йде в ногу з принципом відкритості/закритості: ви зможете розширювати низькорівневі класи і використовувати їх разом з класами бізнес-логіки, не змінюючи код останніх.

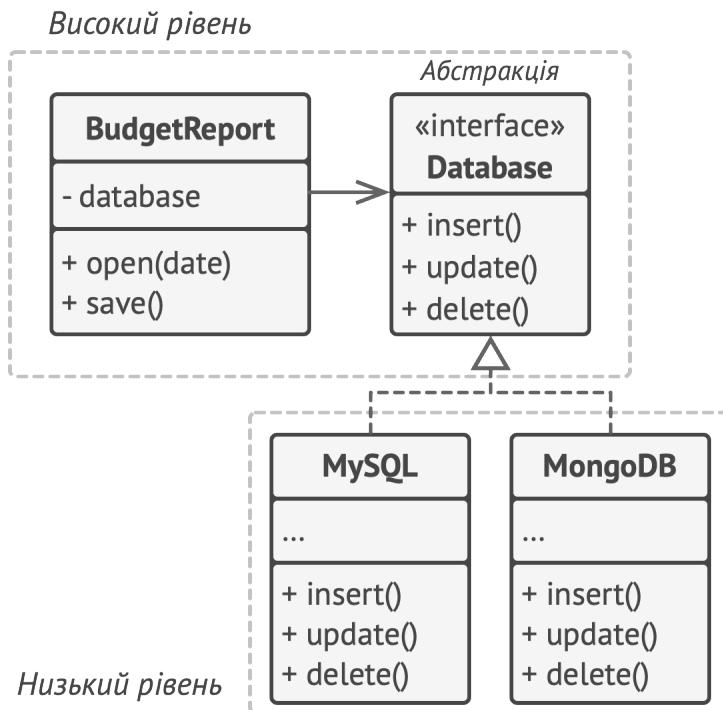
## Приклад

У цьому прикладі високорівневий клас формування бюджетних звітів прямо використовує клас бази даних для завантаження і збереження своєї інформації.



*ДО: високорівневий клас залежить від низькорівневого.*

Ви можете виправити проблему, створивши високорівневий інтерфейс для завантаження/збереження даних і прив'язати до нього клас звітів. Низькорівневі класи повинні реалізовувати цей інтерфейс, щоб їх об'єкти можна було використовувати всередині об'єкта звітів.



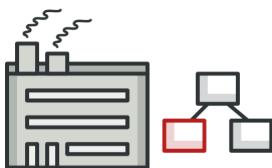
ПІСЛЯ: низькорівневі класи залежать від високорівневої абстракції.

Таким чином, змінюється напрямок залежності. Якщо раніше високий рівень залежав від низького, то зараз все навпаки: низькорівневі класи залежать від високорівневого інтерфейсу.

# КАТАЛОГ ПАТЕРНІВ

# Породжуvalьні патерни

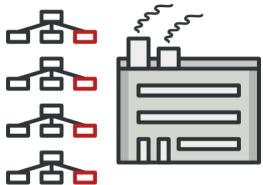
Ці патерни відповідають за зручне та безпечне створення нових об'єктів або навіть цілих сімейств об'єктів.



## Фабричний метод

Factory Method

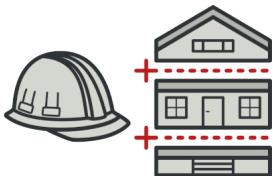
Визначає загальний інтерфейс для створення об'єктів у суперкласі, дозволяючи підкласам змінювати тип створюваних об'єктів.



## Абстрактна фабрика

Abstract Factory

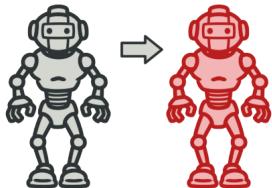
Дає змогу створювати сімейства пов'язаних об'єктів, не прив'язуючись до конкретних класів створюваних об'єктів.



## Будівельник

Builder

Дає змогу створювати складні об'єкти крок за кроком. Будівельник дає можливість використовувати один і той самий код будівництва для отримання різних відображення об'єктів.



## Прототип

Prototype

Дає змогу копіювати об'єкти, не вдаючись у подробиці їхньої реалізації.



## Одинарок

Singleton

Гарантує, що клас має лише один екземпляр, та надає глобальну точку доступу до нього.



# ФАБРИЧНИЙ МЕТОД

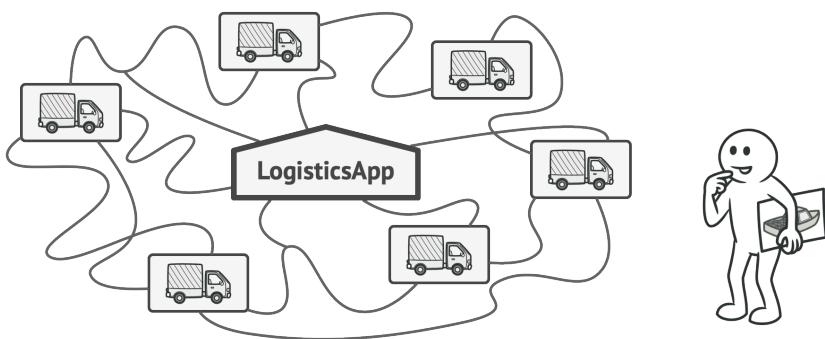
Також відомий як: Віртуальний конструктор, *Factory Method*

**Фабричний метод** – це породжувальний патерн проєктування, який визначає загальний інтерфейс для створення об'єктів у супер класі, дозволяючи під класам змінювати тип створюваних об'єктів.

## (:() Проблема

Уявіть, що ви створюєте програму керування вантажними перевезеннями. Спочатку ви плануєте перевезення товарів тільки вантажними автомобілями. Тому весь ваш код працює з об'єктами класу `Вантажівка`.

Згодом ваша програма стає настільки відомою, що морські перевізники шикуються в чергу і благають додати до програми підтримку морської логістики.



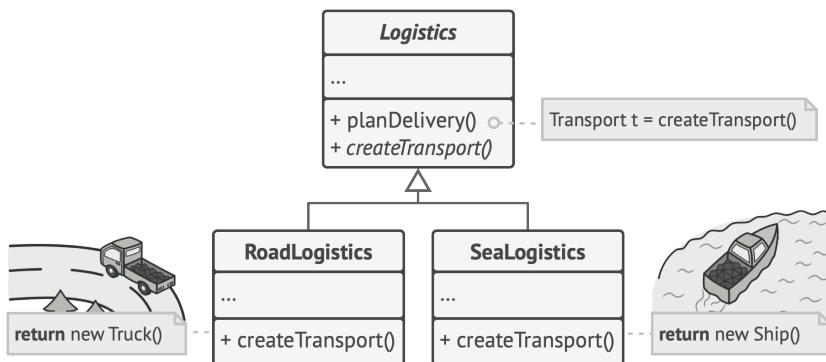
*Додати новий клас не так просто, якщо весь код вже залежить від конкретних класів.*

Чудові новини, чи не так?! Але як щодо коду? Велика частина існуючого коду жорстко прив'язана до класів `Вантажівок`. Щоб додати до програми класи морських `Суден`, знадобиться перелопачувати весь код. Якщо ж ви вирішите додати до програми ще один вид транспорту, тоді всю цю роботу доведеться повторити.

У підсумку ви отримаєте жахливий код, переповнений умовними операторами, що виконують ту чи іншу дію в залежності від вибраного класу транспорту.

## 😊 Рішення

Патерн Фабричний метод пропонує відмовитись від безпосереднього створення об'єктів за допомогою оператора `new`, замінивши його викликом особливого *фабричного* методу. Не лякайтесь, об'єкти все одно будуть створюватися за допомогою `new`, але робити це буде фабричний метод.

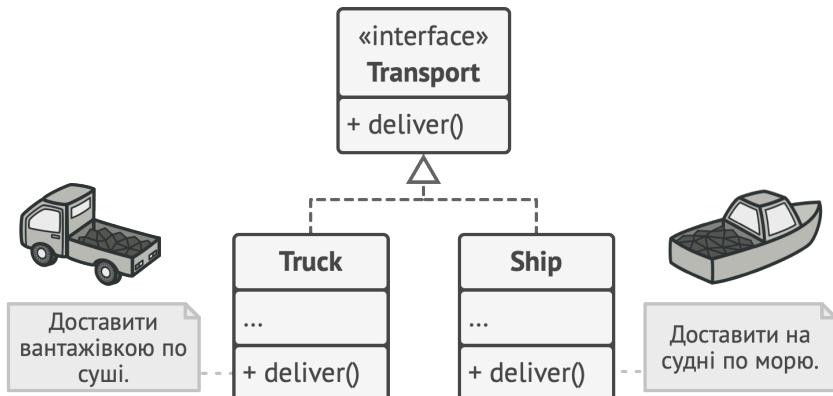


*Підкласи можуть змінювати клас створюваних об'єктів.*

На перший погляд це може здатись безглупдим – ми просто перемістили виклик конструктора з одного кінця програми в інший. Проте тепер ви зможете перевизначити фабричний метод у підкласі, щоб змінити тип створюваного продукту.

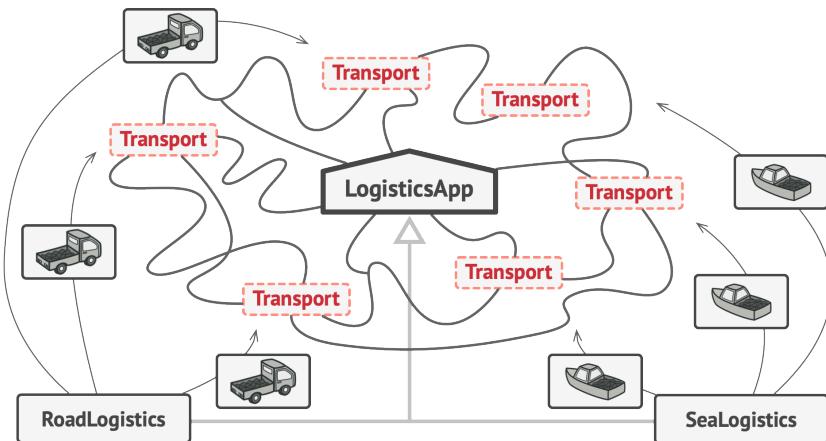
Щоб ця система запрацювала, всі об'єкти, що повертаються, повинні мати спільний інтерфейс. Підкласи зможуть виго-

тovляти об'єкти різних класів, що відповідають одному і тому самому інтерфейсу.



*Vsi ob'єкти-продукти повинні мати спільний інтерфейс.*

Наприклад, класи **Вантажівка** і **Судно** реалізують інтерфейс **Транспорт** з методом **доставити**. Кожен з цих класів реалізує метод по-своєму: вантажівки перевозять вантажі сушою, а судна – морем. Фабричний метод класу **ДорожноЙЛогістики** поверне об'єкт-вантажівку, а класу **МорськоїЛогістики** – об'єкт-судно.

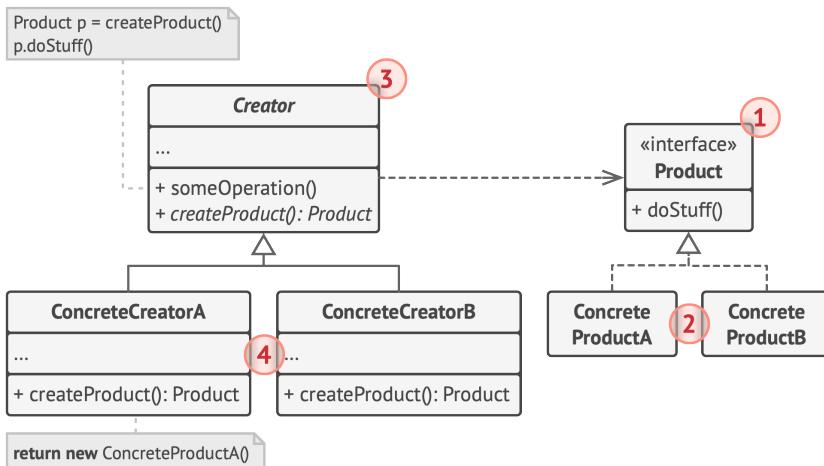


Допоки всі продукти реалізують спільний інтерфейс, їхні об'єкти можна змінювати один на інший у клієнтському коді.

Клієнт фабричного методу не відчує різниці між цими об'єктами, адже він трактуватиме їх як якийсь абстрактний Транспорт .

Для нього буде важливим, щоб об'єкт мав метод доставити , а не те, як конкретно він працює.

## Структура



1. **Продукт** визначає загальний інтерфейс об'єктів, які може створювати творець та його підкласи.
2. **Конкретні продукти** містять код різних продуктів. Продукти відрізняються реалізацією, але інтерфейс у них буде спільним.
3. **Творець** оголошує фабричний метод, який має повертати нові об'єкти продуктів. Важливо, щоб тип результату цього методу співпадав із загальним інтерфейсом продуктів.

Зазвичай, фабричний метод оголошують абстрактним, щоб змусити всі підкласи реалізувати його по-своєму. Однак він може також повертати продукт за замовчуванням.

Незважаючи на назву, важливо розуміти, що створення продуктів **не є** єдиною і головною функцією творця. Зазвичай він містить ще й інший корисний код для роботи з продуктом. Аналогія: у великій софтверній компанії може бути центр підготовки програмістів, але все ж таки основним завданням компанії залишається написання коду, а не навчання програмістів.

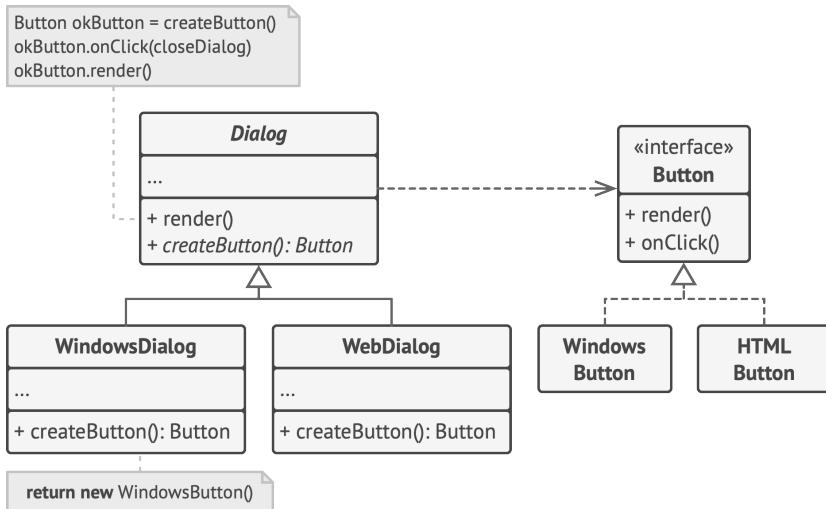
4. **Конкретні творці** по-своєму реалізують фабричний метод, виробляючи ті чи інші конкретні продукти.

Фабричний метод не зобов'язаний створювати нові об'єкти увесь час. Його можна переписати так, аби повернати з яко-гось сховища або кешу вже існуючі об'єкти.

## # Псевдокод

У цьому прикладі **Фабричний метод** допомагає створювати крос-платформові елементи інтерфейсу, не прив'язуючи основний код програми до конкретних класів кожного елементу.

Фабричний метод оголошений у класі діалогів. Його підкласи належать до різних операційних систем. Завдяки фабричному методу, вам не потрібно переписувати логіку діалогів під кожну систему. Підкласи можуть успадковувати майже увесь код базового діалогу, змінюючи типи кнопок та інших елементів, з яких базовий код будує вікна графічного користувачього інтерфейсу.



Приклад крос-платформового діалогу.

Базовий клас діалогів працює з кнопками через їхній загальний програмний інтерфейс. Незалежно від того, яку варіацію кнопок повернув фабричний метод, діалог залишиться робочим. Базовий клас не залежить від конкретних класів кнопок, залишаючи підкласам прийняття рішення про тип кнопок, які необхідно створити.

Такий підхід можна застосувати і для створення інших елементів інтерфейсу. Хоча кожен новий тип елементів наблизитиме вас до **Абстрактної фабрики**.

```

1 // Патерн Фабричний метод має сенс лише тоді, коли в програмі є
2 // ієархія класів продуктів.
3 interface Button is
4   method render()
  
```

```
5  method onClick(f)
6
7  class WindowsButton implements Button is
8    method render(a, b) is
9      // Відобразити кнопку в стилі Windows.
10   method onClick(f) is
11     // Навісити на кнопку обробник подій Windows.
12
13  class HTMLButton implements Button is
14    method render(a, b) is
15      // Повернути HTML-код кнопки.
16    method onClick(f) is
17      // Навісити на кнопку обробник події браузера.
18
19
20 // Базовий клас фабрики. Зауважте, що "фабрика" – це всього лише
21 // додаткова роль для цього класу. Скоріше за все, він вже має
22 // якусь бізнес-логіку, яка потребує створення продуктів.
23 class Dialog is
24   method render() is
25     // Щоб використати фабричний метод, ви маєте
26     // пересвідчитися, що ця бізнес-логіка не залежить від
27     // конкретних класів продуктів. Button – це загальний
28     // інтерфейс кнопок, тому все гаразд.
29   Button okButton = createButton()
30   okButton.onClick(closeDialog)
31   okButton.render()
32
33 // Ми виносимо весь код створення продуктів до особливого
34 // методу, який називають "фабричним".
35 abstract method createButton():Button
36
```

```
37
38 // Конкретні фабрики перевизначають фабричний метод і повертають
39 // з нього власні продукти.
40 class WindowsDialog extends Dialog is
41     method createButton():Button is
42         return new WindowsButton()
43
44 class WebDialog extends Dialog is
45     method createButton():Button is
46         return new HTMLButton()
47
48
49 class Application is
50     field dialog: Dialog
51
52 // Програма створює певну фабрику в залежності від
53 // конфігурації або оточення.
54 method initialize() is
55     config = readApplicationConfigFile()
56
57     if (config.OS == "Windows") then
58         dialog = new WindowsDialog()
59     else if (config.OS == "Web") then
60         dialog = new WebDialog()
61     else
62         throw new Exception("Error! Unknown operating system.")
63
64 // Якщо весь інший клієнтський код працює з фабриками та
65 // продуктами тільки через загальний інтерфейс, то для нього
66 // байдуже, якого типу фабрику було створено на початку.
67 method main() is
68     this.initialize()
```

69 `dialog.render()`

## 💡 Застосування

⚡ Коли типи і залежності об'єктів, з якими повинен працювати ваш код, невідомі заздалегідь.

⚡ Фабричний метод відокремлює код виробництва продуктів від решти коду, який використовує ці продукти.

Завдяки цьому код виробництва можна розширювати, не зачіпаючи основний код. Щоб додати підтримку нового продукту, вам потрібно створити новий підклас та визначити в ньому фабричний метод, повертаючи звідти екземпляр нового продукту.

⚡ Коли ви хочете надати користувачам можливість розширювати частини вашого фреймворку чи бібліотеки.

⚡ Користувачі можуть розширювати класи вашого фреймворку через успадкування. Але як же зробити так, аби фреймворк створював об'єкти цих класів, а не стандартних?

Рішення полягає у тому, щоб надати користувачам можливість розширювати не лише бажані компоненти, але й класи, які їх створюють. Тому ці класи повинні мати конкретні створюючі методи, які можна буде перевизначити.

Наприклад, ви використовуєте готовий UI-фреймворк для свого додатку. Але – от халепа – вам необхідно мати круглі кнопки, а не стандартні прямокутні. Ви створюєте клас `RoundButton`. Але як сказати головному класу фреймворку `UIFramework`, щоб він почав тепер створювати круглі кнопки замість стандартних прямокутних?

Для цього з базового класу фреймворку ви створюєте під- клас `UIWithRoundButtons`, перевизначаєте в ньому метод створення кнопки (а-ля, `createButton`) і вписуєте туди створення свого класу кнопок. Потім використовуєте `UIWithRoundButtons` замість стандартного `UIFramework`.

 **Коли ви хочете зекономити системні ресурси, повторно використовуючи вже створені об'єкти, замість породження нових.**

 Така проблема зазвичай виникає під час роботи з «важкими», вимогливими до ресурсів об'єктами, такими, як підключення до бази даних, файлової системи й подібними.

Уявіть, скільки дій вам потрібно зробити, аби повторно використовувати вже існуючі об'єкти:

1. Спочатку слід створити загальне сховище, щоб зберігати в ньому всі створювані об'єкти.
2. При запиті нового об'єкта потрібно буде подивитись у сховище та перевірити, чи є там невикористаний об'єкт.

3. Потім повернути його клієнтському коду.
4. Але якщо ж вільних об'єктів немає, створити новий, не забувши додати його до сховища.

Увесь цей код потрібно десь розмістити, щоб не засмічувати клієнтський код.

Найзручнішим місцем був би конструктор об'єкта, адже всі ці перевірки потрібні тільки під час створення об'єктів, але, на жаль, конструктор завжди створює **нові** об'єкти, тому він не може повернути існуючий екземпляр.

Отже, має бути інший метод, який би віддавав як існуючі, так і нові об'єкти. Ним і стане фабричний метод.

## Кроки реалізації

1. Приведіть усі створювані продукти до загального інтерфейсу.
2. Створіть порожній фабричний метод у класі, який виробляє продукти. В якості типу, що повертається, вкажіть загальний інтерфейс продукту.
3. Пройдіться по коду класу й знайдіть усі ділянки, що створюють продукти. По черзі замініть ці ділянки викликами фабричного методу, переносячи в нього код створення різних продуктів.

Можливо, доведеться додати до фабричного методу декілька параметрів, що контролюють, який з продуктів потрібно створити.

Імовірніше за все, фабричний метод виглядатиме гнітюче на цьому етапі. В ньому житиме великий умовний оператор, який вибирає клас створюваного продукту. Але не хвилюйтесь, ми ось-ось все це віпправимо.

4. Для кожного типу продуктів заведіть підклас і перевизначте в ньому фабричний метод. З суперкласу перемістіть туди код створення відповідного продукту.
5. Якщо створюваних продуктів занадто багато для існуючих підкласів творця, ви можете подумати про введення параметрів до фабричного методу, аби повернати різні продукти в межах одного підкласу.

Наприклад, у вас є клас `Пошта` з підкласами `Авіапошта` і `НаземнаПошта`, а також класи продуктів `Літак`, `Вантажівка` й `Потяг`. `Авіа` відповідає `Літакам`, але для `НаземноїПошти` є відразу два продукти. Ви могли б створити новий підклас пошти й для потягів, але проблему можна вирішити по-іншому. Клієнтський код може передавати до фабричного методу `НаземноїПошти` аргумент, що контролює, який з продуктів буде створено.

6. Якщо після цих всіх переміщень фабричний метод став порожнім, можете зробити його абстрактним. Якщо ж у

ньому щось залишилося — не страшно, це буде його типовою реалізацією (за замовчуванням).

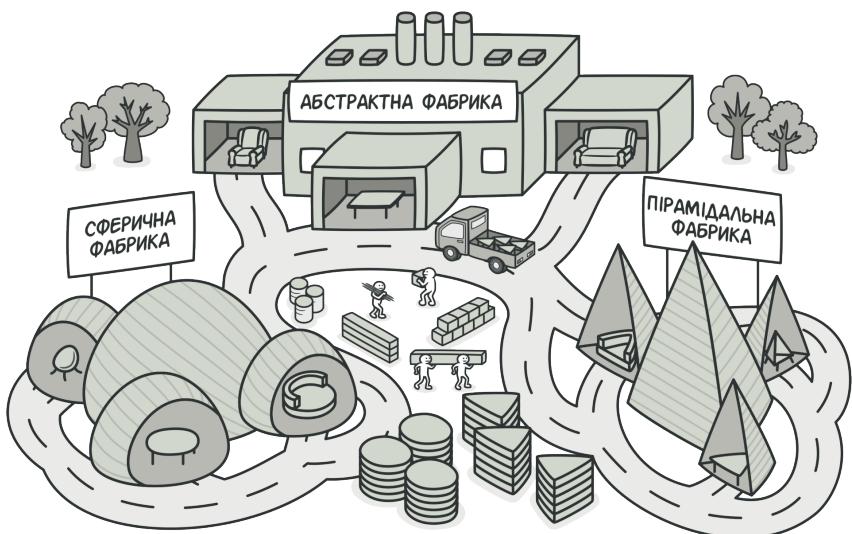
## ΔΔ Переваги та недоліки

- ✓ Позбавляє клас від прив'язки до конкретних класів продуктів.
- ✓ Виділяє код виробництва продуктів в одне місце, спрощуючи підтримку коду.
- ✓ Спрошує додавання нових продуктів до програми.
- ✓ Реалізує *принцип відкритості/закритості*.
- ✗ Може привести до створення великих паралельних ієархій класів, адже для кожного класу продукту потрібно створити власний підклас творця.

## ↔ Відносини з іншими патернами

- Багато архітектур починаються із застосування Фабричного методу (простішого та більш розширюваного за допомогою підкласів) та еволюціонують у бік Абстрактної фабрики, Прототипу або Будівельника (гнучкіших, але й складніших).
- Класи Абстрактної фабрики найчастіше реалізуються за допомогою Фабричного методу, хоча вони можуть бути побудовані і на основі Прототипу.

- **Фабричний метод** можна використовувати разом з **Ітератором**, щоб підкласи колекцій могли створювати необхідні їм ітератори.
- **Прототип** не спирається на спадкування, але йому потрібна складна операція ініціалізації. **Фабричний метод**, навпаки, побудований на спадкуванні, але не вимагає складної ініціалізації.
- **Фабричний метод** можна розглядати як окремий випадок **Шаблонного методу**. Крім того, *Фабричний метод* нерідко буває частиною великого класу з *Шаблонними методами*.



# АБСТРАКТНА ФАБРИКА

Також відомий як: *Abstract Factory*

**Абстрактна фабрика** – це породжувальний патерн проєктування, що дає змогу створювати сімейства пов'язаних об'єктів, не прив'язуючись до конкретних класів створюваних об'єктів.

## (:() Проблема

Уявіть, що ви пишете симулятор меблевого магазину. Ваш код містить:

1. Сімейство залежних продуктів. Скажімо, Крісло + Диван + Столик .
2. Кілька варіацій цього сімейства. Наприклад, продукти Крісло , Диван та Столик представлені в трьох різних стилях: Ар-деко , Вікторіанському і Модерн .



*Сімейства продуктів та їхніх варіацій.*

Вам потрібно створювати об'єкти продуктів у такий спосіб, щоб вони завжди пасували до інших продуктів того само-

го сімейства. Це дуже важливо, адже клієнти засмучуються, коли отримують меблі, що не можна поєднати між собою.

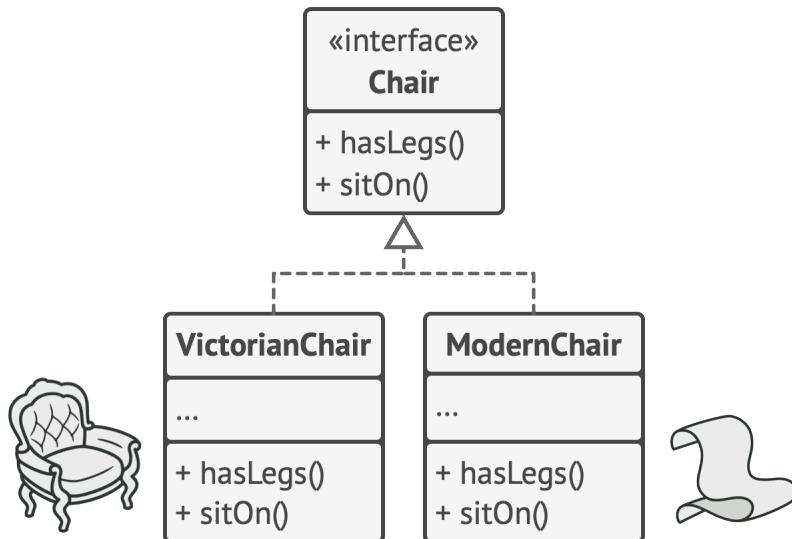


*Клієнти засмучуються, якщо отримують продукти, що не поєднуються.*

Крім того, ви не хочете вносити зміни в існуючий код під час додавання в програму нових продуктів або сімейств. Постачальники часто оновлюють свої каталоги, але ви б не хотіли змінювати вже написаний код кожен раз при надходженні нових моделей меблів.

## 😊 Рішення

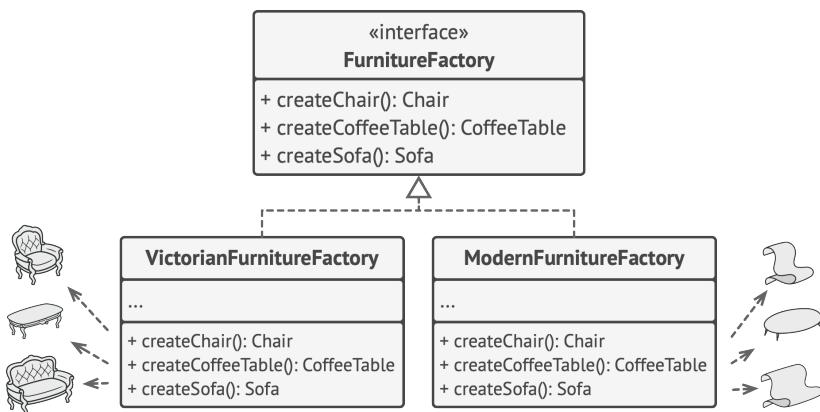
Для початку, патерн Абстрактна фабрика пропонує виділити загальні інтерфейси для окремих продуктів, що складають одне сімейство, і описати в них спільну для цих продуктів поведінку. Так, наприклад, усі варіації крісел отримають спільний інтерфейс `Крісло`, усі дивани реалізують інтерфейс `Диван` тощо.



*Всі варіації одного й того самого об'єкта мають жити в одній ієрархії класів.*

Далі ви створюєте *абстрактну фабрику* – загальний інтерфейс, який містить методи створення всіх продуктів сімейства (наприклад, `створитиКрісло`, `створитиДиван` і `створитиСтолик`). Ці операції повинні повернати **абстрактні** типи продуктів, представлені інтерфейсами, які ми виділили раніше – `Крісла`, `Дивани` і `Столики`.

Як щодо варіацій продуктів? Для кожної варіації сімейства продуктів ми повинні створити свою власну фабрику, реалізувавши абстрактний інтерфейс. Фабрики створюють продукти однієї варіації. Наприклад, `ФабрикаМодерн` буде повервати тільки `КріслаМодерн`, `ДиваниМодерн` і `СтоликиМодерн`.



Конкретні фабрики відповідають певній варіації сімейства продуктів.

Клієнтський код повинен працювати як із фабриками, так і з продуктами тільки через їхні загальні інтерфейси. Це дозволить подавати у ваші класи будь-які типи фабрик і виробляти будь-які типи продуктів, без необхідності вносити зміни в існуючий код.

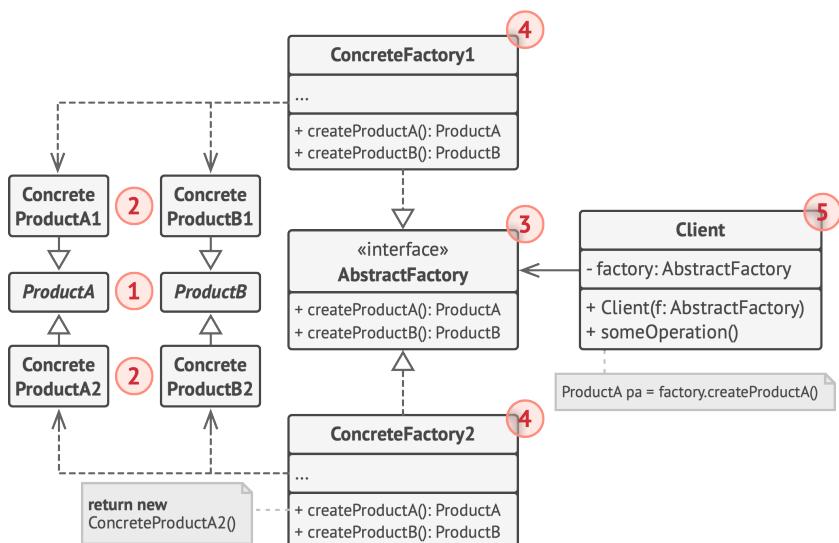


Для клієнтського коду повинно бути не важливо, з якою фабрикою працювати.

Наприклад, клієнтський код просить фабрику зробити стілець. Він не знає, якому типу відповідає ця фабрика. Він не знає, отримає вікторіанський або модерновий стілець. Для нього важливо, щоб на цьому стільці можна було сидіти та щоб цей стілець відмінно виглядав поруч із диваном тієї ж фабрики.

Залишилося прояснити останній момент: хто ж створює об'єкти конкретних фабрик, якщо клієнтський код працює лише із загальними інтерфейсами? Зазвичай програма створює конкретний об'єкт фабрики під час запуску, причому тип фабрики вибирається на підставі параметрів оточення або конфігурації.

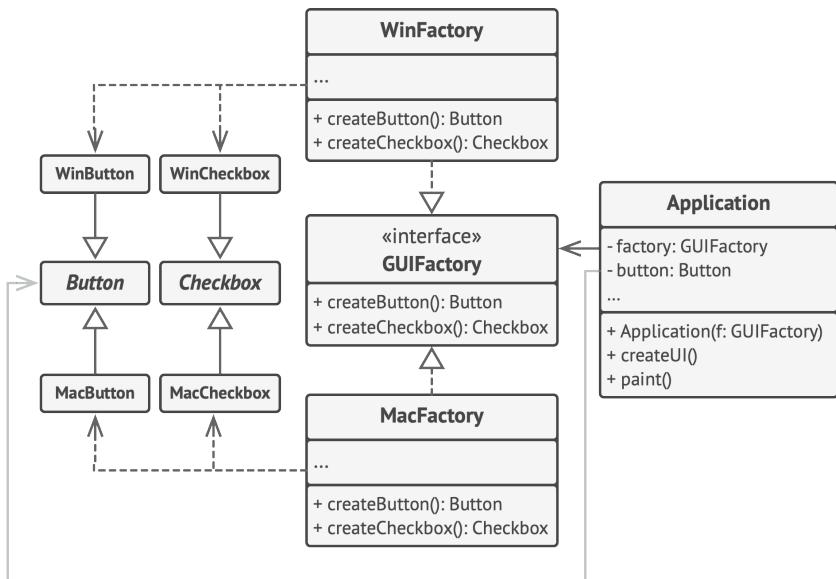
## Структура



1. **Абстрактні продукти** оголошують інтерфейси продуктів, що пов'язані один з одним за змістом, але виконують різні функції.
2. **Конкретні продукти** – великий набір класів, що належать до різних абстрактних продуктів (крісло/столик), але мають одні й ті самі варіації (Вікторіанський/Модерн).
3. **Абстрактна фабрика** оголошує методи створення різних абстрактних продуктів (крісло/столик).
4. **Конкретні фабрики** кожна належить до своєї варіації продуктів (Вікторіанський/Модерн) і реалізує методи абстрактної фабрики, даючи змогу створювати всі продукти певної варіації.
5. Незважаючи на те, що конкретні фабрики породжують конкретні продукти, сигнатури їхніх методів мусять повертати відповідні абстрактні продукти. Це дозволить клієнтського коду, що використовує фабрику, не прив'язуватися до конкретних класів продуктів. Клієнт зможе працювати з будь-якими варіаціями продуктів через абстрактні інтерфейси.

## # Псевдокод

У цьому прикладі **Абстрактна фабрика** створює крос-платформові елементи інтерфейсу і стежить за тим, щоб вони відповідали обраній операційній системі.



*Приклад крос-платформового графічного інтерфейсу користувача.*

Крос-платформова програма може відображати одні й ті самі елементи інтерфейсу по-різному, в залежності від обраної операційної системи. Важливо, щоб у такій програмі всі створювані елементи завжди відповідали поточній операційній системі. Ви ж не хотіли б, аби програма, запущена на Windows, раптом почала показувати чек-бокси в стилі macOS?

Абстрактна фабрика оголошує список створюючих методів, які клієнтський код може використовувати для отримання тих чи інших різновидів елементів інтерфейсу. Конкретні фабрики відносяться до різних операційних систем і створюють елементи, сумісні з цією системою.

Програма на самому початку визначає фабрику, що відповідає поточній операційній системі. Потім створює цю фабрику та віддає її клієнтському коду. У подальшому, щоб виключити несумісність продуктів, що повертаються, клієнт працюватиме тільки з цією фабрикою.

Клієнтський код не залежить від конкретних класів фабрик чи елементів інтерфейсу. Він спілкується з ними через загальні інтерфейси, не залежачи від конкретних класів фабрик чи елементів користувачького інтерфейсу.

Таким чином, щоб додати до програми нову варіацію елементів інтерфейсу (наприклад, для підтримки Linux), вам не потрібно змінювати клієнтський код. Достатньо створити ще одну фабрику, що виготовляє ці елементи.

```

1 // Цей патерн передбачає, що ви маєте кілька сімейств продуктів,
2 // які знаходяться в окремих ієархіях класів (Button/Checkbox).
3 // Продукти одного сімейства повинні мати спільний інтерфейс.
4 interface Button is
5     method paint()
6
7     // Сімейства продуктів мають однакові варіації (macOS/Windows).
8 class WinButton implements Button is
9     method paint() is
10        // Відобразити кнопку в стилі Windows.
11
12 class MacButton implements Button is
13     method paint() is
14        // Відобразити кнопку в стилі macOS.

```

```
15 interface Checkbox is
16     method paint()
17
18 class WinCheckbox implements Checkbox is
19     method paint() is
20         // Відобразити чекбокс в стилі Windows.
21
22 class MacCheckbox implements Checkbox is
23     method paint() is
24         // Відобразити чекбокс в стилі macOS.
25
26
27 // Абстрактна фабрика знає про всі абстрактні типи продуктів.
28 interface GUIFactory is
29     method createButton():Button
30     method createCheckbox():Checkbox
31
32 // Кожна конкретна фабрика знає лише про продукти своєї варіації
33 // і створює лише їх.
34 class WinFactory implements GUIFactory is
35     method createButton():Button is
36         return new WinButton()
37     method createCheckbox():Checkbox is
38         return new WinCheckbox()
39
40 // Незважаючи на те, що фабрики оперують конкретними класами,
41 // їхні методи повертають абстрактні типи продуктів. Завдяки
42 // цьому фабрики можна замінити одну на іншу, не змінюючи
43 // клієнтського коду.
44 class MacFactory implements GUIFactory is
45     method createButton():Button is
46         return new MacButton()
```

```
47 method createCheckbox(): Checkbox is
48     return new MacCheckbox()
49
50
51 // Для коду, який використовує фабрику, не важливо, з якою
52 // конкретно фабрикою він працює. Всі отримувачі продуктів
53 // працюють з ними через загальні інтерфейси.
54 class Application is
55     private field factory: GUIFactory
56     private field button: Button
57     constructor Application(factory: GUIFactory) is
58         this.factory = factory
59     method createUI()
60         this.button = factory.createButton()
61     method paint()
62         button.paint()
63
64
65 // Програма вибирає тип конкретної фабрики й створює її
66 // динамічно, виходячи з конфігурації або оточення.
67 class ApplicationConfigurator is
68     method main() is
69         config = readApplicationConfigFile()
70
71         if (config.OS == "Windows") then
72             factory = new WinFactory()
73         else if (config.OS == "Mac") then
74             factory = new MacFactory()
75         else
76             throw new Exception("Error! Unknown operating system.")
77
78         Application app = new Application(factory)
```

## Застосування

-  **Коли бізнес-логіка програми повинна працювати з різними видами пов'язаних один з одним продуктів, незалежно від конкретних класів продуктів.**
-  Абстрактна фабрика приховує від клієнтського коду подробиці того, як і які конкретно об'єкти будуть створені. Внаслідок цього, клієнтський код може працювати з усіма типами створюваних продуктів, так як їхній загальний інтерфейс був визначений заздалегідь.
-  **Коли в програмі вже використовується Фабричний метод, але чергові зміни передбачають введення нових типів продуктів.**
-  У будь-якій добротній програмі кожен клас *має відповідати лише за одну річ*. Якщо клас має занадто багато фабричних методів, вони здатні затуманити його основну функцію. Тому є сенс у тому, щоб винести усю логіку створення продуктів в окрему ієрархію класів, застосувавши абстрактну фабрику.

## Кроки реалізації

1. Створіть таблицю спiввiдношень типiв продуктiв до варiацiй сiмейств продуктiв.
2. Зведiть усi варiацiї продуктiв до загальних iнтерфейсiв.

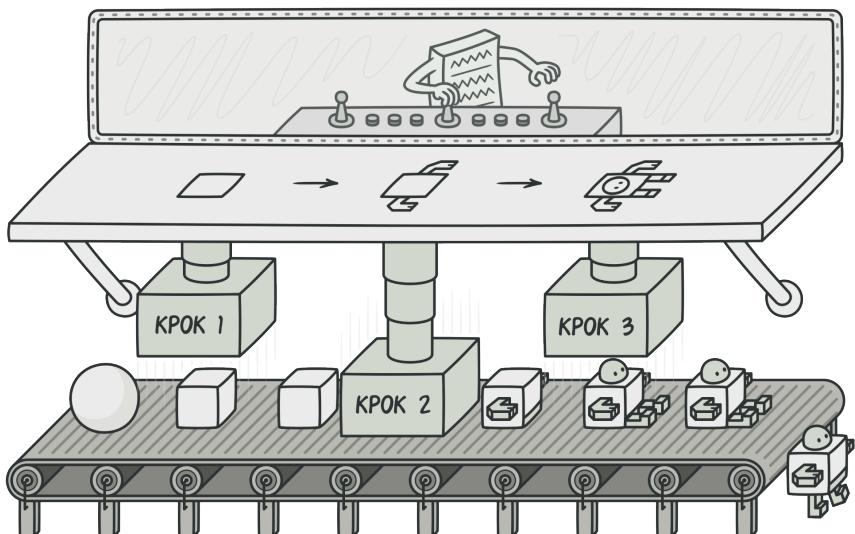
3. Визначте інтерфейс абстрактної фабрики. Він повинен мати фабричні методи для створення кожного типу продуктів.
4. Створіть класи конкретних фабрик, реалізувавши інтерфейс абстрактної фабрики. Цих класів має бути стільки ж, скільки й варіацій сімейств продуктів.
5. Змініть код ініціалізації програми так, щоб вона створювала певну фабрику й передавала її до клієнтського коду.
6. Замініть у клієнтському коді ділянки створення продуктів через конструктор на виклики відповідних методів фабрики.

## Переваги та недоліки

- ✓ Гарантує поєднання створюваних продуктів.
- ✓ Звільняє клієнтський код від прив'язки до конкретних класів продукту.
- ✓ Виділяє код виробництва продуктів в одне місце, спрощуючи підтримку коду.
- ✓ Спрощує додавання нових продуктів до програми.
- ✓ Реалізує *принцип відкритості/закритості*.
- ✗ Ускладнює код програми внаслідок введення великої кількості додаткових класів.
- ✗ Вимагає наявності всіх типів продукту в кожній варіації.

## ↔ Відносини з іншими патернами

- Багато архітектур починаються із застосування Фабричного методу (простішого та більш розширеного за допомогою підкласів) та еволюціонують у бік Абстрактної фабрики, Прототипу або Будівельника (гнучкіших, але й складніших).
- Будівельник концентрується на будівництві складних об'єктів крок за кроком. Абстрактна фабрика спеціалізується на створенні сімейств пов'язаних продуктів. Будівельник повертає продукт тільки після виконання всіх кроків, а Абстрактна фабрика повертає продукт одразу.
- Класи Абстрактної фабрики найчастіше реалізуються за допомогою Фабричного методу, хоча вони можуть бути побудовані і на основі Прототипу.
- Абстрактна фабрика може бути використана замість Фасаду для того, щоб приховати платформо-залежні класи.
- Абстрактна фабрика може працювати спільно з Мостом. Це особливо корисно, якщо у вас є абстракції, які можуть працювати тільки з деякими реалізаціями. В цьому випадку фабрика визначатиме типи створюваних абстракцій та реалізацій.
- Абстрактна фабрика, Будівельник та Прототип можуть реалізовуватися за допомогою Одинака.



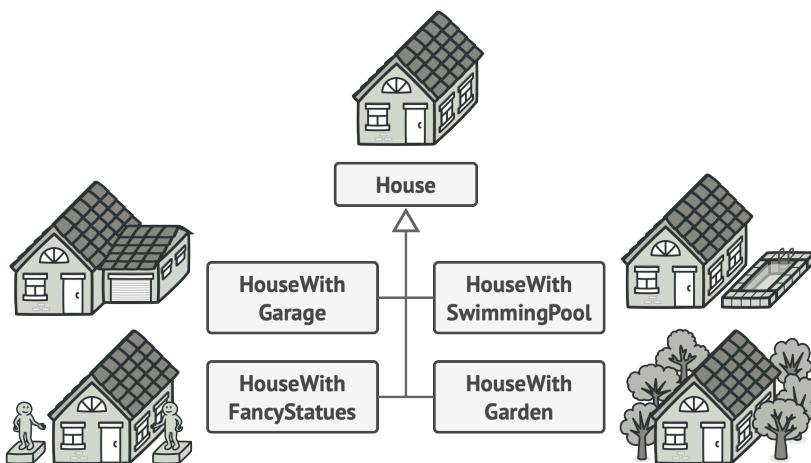
# БУДІВЕЛЬНИК

Також відомий як: *Builder*

**Будівельник** – це породжуvalьний патерн проектування, що дає змогу створювати складні об'єкти крок за кроком. Будівельник дає можливість використовувати один і той самий код будівництва для отримання різних відображеній об'єктів.

## (:() Проблема

Уявіть складний об'єкт, що вимагає кропіткої покрокової ініціалізації безлічі полів і вкладених об'єктів. Код ініціалізації таких об'єктів зазвичай захований всередині монстроподібного конструктора з десятком параметрів. Або ще – розпорощений по всьому клієнтському коду.



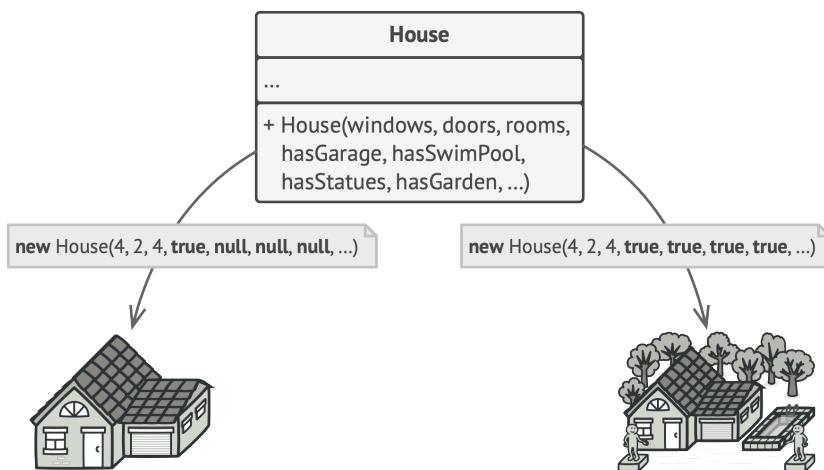
*Створивши купу підкласів для всіх конфігурацій об'єктів, ви можете надміру ускладнити програму.*

Наприклад, подумаймо про те, як створити об'єкт `Будинок`. Щоб побудувати стандартний будинок, потрібно: звести 4 стіни, встановити двері, вставити пару вікон та постелити дах. Але що робити, якщо ви хочете більший та світліший будинок, що має басейн, сад та інше добро?

Найпростіше рішення – розширити клас `Будинок`, створивши підкласи для всіх комбінацій параметрів будинку.

Проблема такого підходу – величезна кількість класів, які вам доведеться створити. Кожен новий параметр, на кшталт кольору шпалер чи матеріалу покрівлі, змусить вас створювати все більше й більше класів для перерахування усіх можливих варіантів.

Аби не плодити підкласи, можна підійти до вирішення питання з іншого боку. Ви можете створити гіантський конструктор **Будинку**, що приймає безліч параметрів для контролю над створюваним продуктом. Так, це позбавить вас від підкласів, але призведе до появи іншої проблеми.



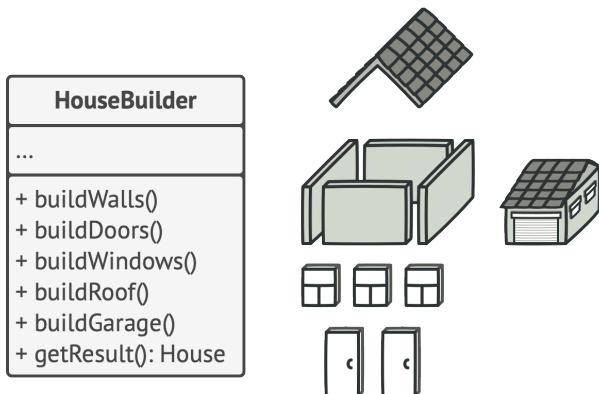
*Конструктор з безліччю параметрів має свій недолік: не всі параметри потрібні протягом більшої частини часу.*

Більшість цих параметрів буде простоювати, а виклики конструктора будуть виглядати монстроподібно через **довгий список параметрів**. Наприклад, басейн є далеко не в кожно-

му будинку, тому параметри, пов'язані з басейнами, даремно простоюватимуть у 99% випадків.

## 😊 Рішення

Патерн Будівельник пропонує внести конструювання об'єкта за межі його власного класу, доручивши цю справу окремим об'єктам, які називаються *будівельниками*.



*Будівельник дозволяє створювати складні об'єкти покроково. Проміжний результат захищений від стороннього втручання.*

Патерн пропонує розбити процес конструювання об'єкта на окремі кроки (наприклад, побудувати Стіни, встановити Двері і т. д.) Щоб створити об'єкт, вам потрібно по черзі викликати методи будівельника. До того ж не потрібно викликати всі кроки, а лише ті, що необхідні для виробництва об'єкта певної конфігурації.

Зазвичай один і той самий крок будівництва може відрізнятися для різних варіацій виготовлених об'єктів. Наприклад, дерев'яний будинок потребує будівництва стін з дерева, а кам'яний – з каменю.

У цьому випадку ви можете створити кілька класів будівельників, які по-різному виконуватимуть ті ж самі кроки. Використовуючи цих будівельників в одному й тому самому будівельному процесі, ви зможете отримувати на виході різні об'єкти.



*Різні будівельники виконують одне і те саме завдання по-різному.*

Наприклад, один будівельник робить стіни з дерева і скла, інший – з каменю і заліза, третій – із золота та діамантів. Викликавши одні й ті самі кроки будівництва, у першому випадку ви отримаєте звичайний житловий будинок, у другому – маленьку фортецю, а в третьому – розкішне житло. Зауважу, що код, який викликає крохи будівництва, пови-

нен працювати з будівельниками через загальний інтерфейс, щоб їх можна було вільно замінювати один на інший.

## Директор

Ви можете піти далі та виділити виклики методів будівельника в окремий клас, що називається «Директором». У цьому випадку директор задаватиме порядок кроків будівництва, а будівельник – виконуватиме їх.

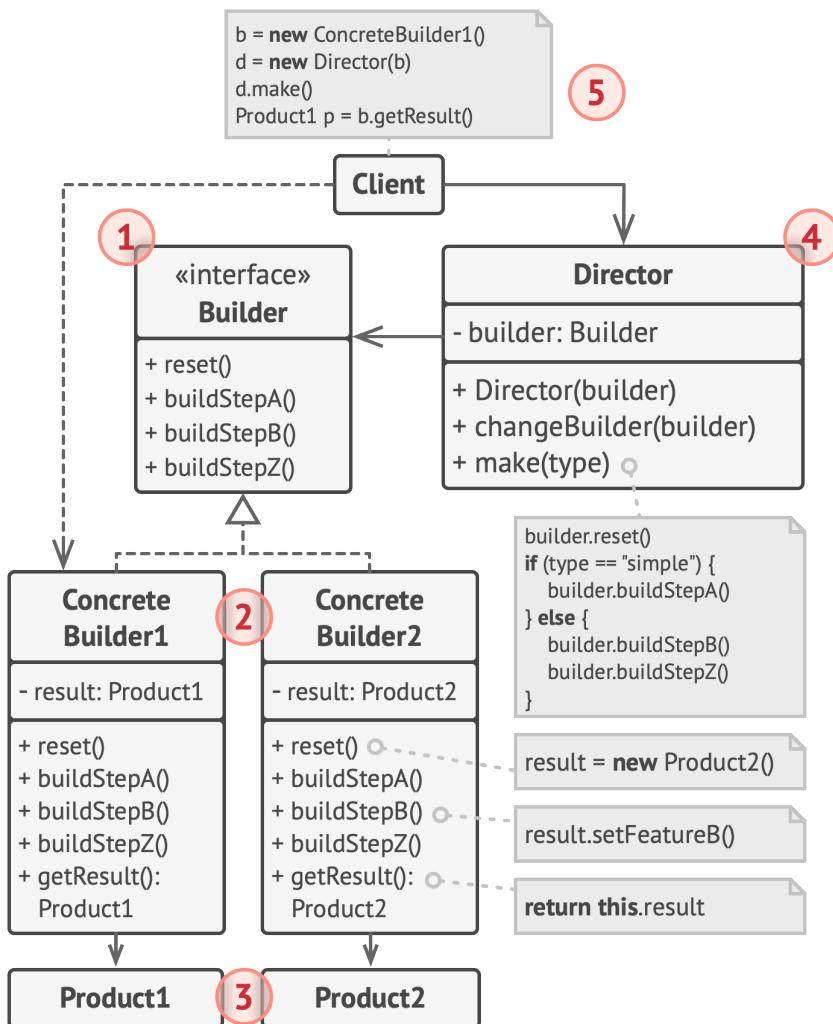


*Директор знає, які кроки повинен виконати об'єкт-будівельник, щоб виготовити продукт.*

Окремий клас *директора* не є суvero обов'язковим. Ви можете викликати методи будівельника і безпосередньо з клієнтського коду. Тим не менш, директор корисний, якщо у вас є кілька способів конструювання продуктів, що відрізняються порядком і наявними кроками конструювання. У цьому випадку ви зможете об'єднати всю цю логіку в одному класі.

Така структура класів повністю приховає від клієнтського коду процес конструювання об'єктів. Клієнту залишиться лише прив'язати бажаного будівельника до директора, а потім отримати від будівельника готовий результат.

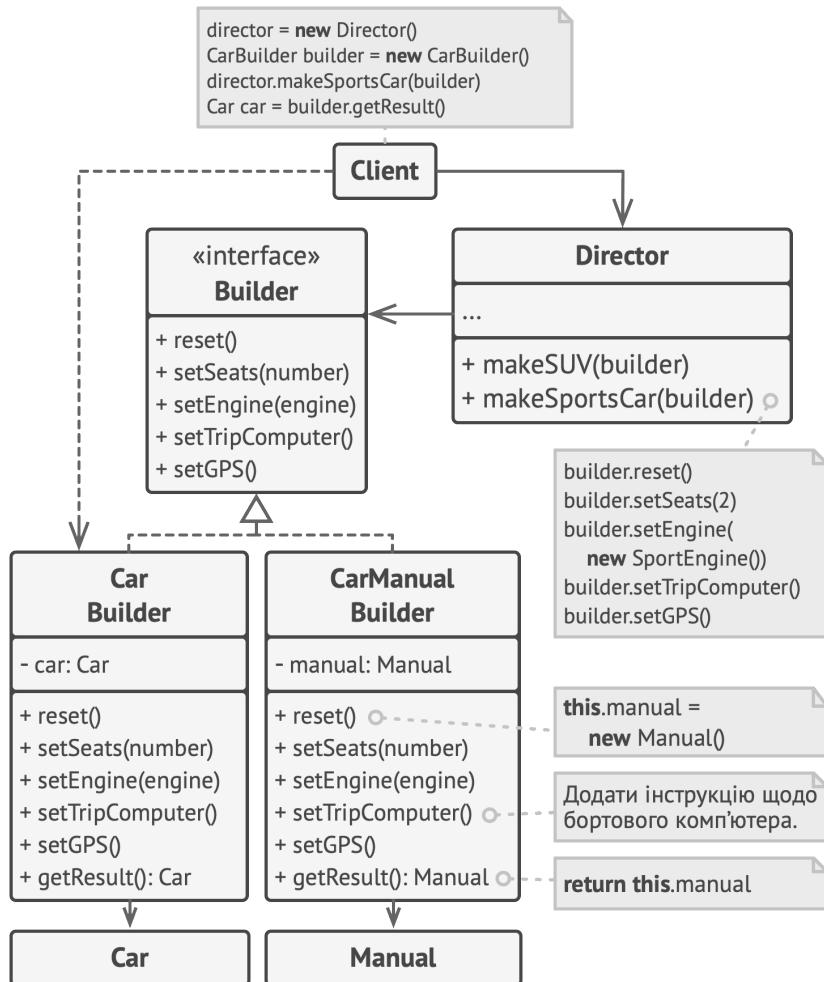
## Структура



1. **Інтерфейс будівельника** оголошує кроки конструювання продуктів, спільні для всіх видів будівельників.
2. **Конкретні будівельники** реалізують кроки будівництва, кожен по-своєму. Конкретні будівельники можуть виготовляти різноманітні об'єкти, що не мають спільного інтерфейсу.
3. **Продукт** – об'єкт, що створюється. Продукти, зроблені різними будівельниками, не зобов'язані мати спільний інтерфейс.
4. **Директор** визначає порядок виклику кроків будівельників, необхідних для виробництва продуктів тієї чи іншої конфігурації.
5. Зазвичай **Клієнт** подає до конструктора директора вже готовий об'єкт-будівельник, а директор надалі використовує тільки його. Але можливим є також інший варіант, коли клієнт передає будівельника через параметр будівельного методу директора. У такому випадку можна щоразу використовувати різних будівельників для виробництва різноманітних відображеній об'єктів.

## # Псевдокод

У цьому прикладі **Будівельник** використовується для покровового конструювання автомобілів та технічних посібників до них.



Приклад покрокового конструювання автомобілів та інструкцій до них.

Автомобіль – це складний об'єкт, який можна налаштувати сотнею різних способів. Замість того, щоб налаштовувати автомобіль через конструктор, ми внесемо його збирання в окремий клас-будівельник, передбачивши методи для конфігурації всіх частин автомобіля.

Клієнт може збирати автомобілі, працюючи з будівельником безпосередньо. З іншого боку, він може доручити цю справу директору. Це об'єкт, який знає, які кроки будівельника потрібно викликати, щоб отримати кілька найпопулярніших конфігурацій автомобілів.

Проте, до кожного автомобіля ще потрібен посібник користувача, що відповідає його конфігурації. Для цього ми створимо ще один клас будівельника, який замість конструювання автомобіля друкуватиме сторінки посібника до тієї деталі, яку ми вбудовуємо в продукт. Тепер, пропустивши через одні й ті самі кроки обидва типи будівельників, ми отримаємо автомобіль та відповідний до нього посібник користувача.

Очевидно, що паперовий посібник і металевий автомобіль – це дві абсолютно різні речі. З цієї причини ми повинні отримувати результат безпосередньо від будівельників, а не від директора. Інакше нам довелося б жорстко прив'язати директора до конкретних класів автомобілів і посібників.

```

1 // Будівельник може створювати різні продукти, використовуючи
2 // один і той самий процес будівництва.
3 class Car is
4     // Автомобілі можуть відрізнятися комплектацією: типом
5     // двигуна, кількістю сидінь, можуть мати або не мати GPS і
6     // систему навігації тощо. Крім того, автомобілі можуть бути
7     // міськими, спортивними або позашляховиками.
8

```

```
9  class Manual is
10 // Посібник користувача для даної конфігурації автомобіля.
11
12
13 // Інтерфейс будівельників оголошує всі можливі етапи та кроки
14 // конфігурації продукту.
15 interface Builder is
16     method reset()
17     method setSeats(...)
18     method setEngine(...)
19     method setTripComputer(...)
20     method setGPS(...)
21
22 // Усі конкретні будівельники реалізують загальний інтерфейс по-
23 // своєму.
24 class CarBuilder implements Builder is
25     private field car:Car
26     method reset()
27         // Помістити новий об'єкт Car в полі "car".
28     method setSeats(...) is
29         // Встановити вказану кількість сидінь.
30     method setEngine(...) is
31         // Встановити наданий двигун.
32     method setTripComputer(...) is
33         // Встановити надану систему навігації.
34     method setGPS(...) is
35         // Встановити або зняти GPS.
36     method getResult(): Car is
37         // Повернути поточний об'єкт автомобіля.
38
39 // На відміну від інших породжуvalьних патернів, де продукти
40 // мають бути частиною однієї ієрархії класів або слідувати
```

```
41 // загальному інтерфейсу, будівельники можуть створювати
42 // абсолютно різні продукти, які не мають спільного предка.
43 class CarManualBuilder implements Builder is
44     private field manual:Manual
45     method reset()
46         // Помістити новий об'єкт Manual у полі "manual".
47     method setSeats(...) is
48         // Описати кількість місць в автівці.
49     method setEngine(...) is
50         // Додати до посібника опис двигуна.
51     method setTripComputer(...) is
52         // Додати до посібника опис системи навігації.
53     method setGPS(...) is
54         // Додати до посібника інструкцію для GPS.
55     method getResult(): Manual is
56         // Повернути поточний об'єкт посібника.
57
58
59 // Директор знає, в якій послідовності потрібно змушувати
60 // працювати будівельника, щоб отримати ту чи іншу версію
61 // продукту. Зауважте, що директор працює з будівельником через
62 // загальний інтерфейс, завдяки чому він не знає тип продукту,
63 // який виготовляє будівельник.
64 class Director is
65     method constructSportsCar(builder: Builder) is
66         builder.reset()
67         builder.setSeats(2)
68         builder.setEngine(new SportEngine())
69         builder.setTripComputer(true)
70         builder.setGPS(true)
71
72
```

```

73 // Директор отримує об'єкт конкретного будівельника від клієнта
74 // (програми). Програма сама знає, якого будівельника
75 // використати, аби отримати потрібний продукт.
76 class Application is
77     method makeCar() is
78         director = new Director()
79
80         CarBuilder builder = new CarBuilder()
81         director.constructSportsCar(builder)
82         Car car = builder.getResult()
83
84         CarManualBuilder builder = new CarManualBuilder()
85         director.constructSportsCar(builder)
86
87         // Готовий продукт повертає будівельник, оскільки
88         // директор частіше за все не знає і не залежить від
89         // конкретних класів будівельників та продуктів.
90         Manual manual = builder.getResult()

```

## 💡 Застосування

 Коли ви хочете позбутися від «телескопічного конструктора».

 Припустімо, у вас є один конструктор з десятьма опціональними параметрами. Його незручно викликати, тому ви створили ще десять конструкторів з меншою кількістю параметрів. Все, що вони роблять, – це переадресовують виклик до базового конструктора, подаючи якісь типові значення в параметри, які відсутні в них самих.

```

1 class Pizza {
2     Pizza(int size) { ... }
3     Pizza(int size, boolean cheese) { ... }
4     Pizza(int size, boolean cheese, boolean pepperoni) { ... }
5     // ...

```

*Такого монстра можна створити тільки в мовах, що мають механізм перевантаження методів, наприклад, C# або Java.*

Патерн Будівельник дозволяє збирати об'єкти покроково, викликаючи тільки ті кроки, які вам потрібні. Отже, більше не потрібно намагатися «запхати» до конструктора всі можливі опції продукту.

 **Коли ваш код повинен створювати різні уявлення якогось об'єкта. Наприклад, дерев'яні та залізобетонні будинки.**

 Будівельник можна застосувати, якщо створення кількох відображенень об'єкта складається з однакових етапів, які відрізняються деталями.

Інтерфейс будівельників визначить всі можливі етапи конструювання. Кожному відображенню відповідатиме власний клас-будівельник. Порядок етапів будівництва визначатиме клас-директор.

 **Коли вам потрібно збирати складні об'єкти, наприклад, дерева Компонувальника.**

 Будівельник конструює об'єкти покроково, а не за один прохід. Більш того, кроки будівництва можна виконувати рекурсивно. А без цього не побудувати деревоподібну структуру на зразок **Компонувальника**.

Зауважте, що Будівельник не дозволяє стороннім об'єктам отримувати доступ до об'єкта, що конструюється, доки той не буде повністю готовий. Це захищає клієнтський код від отримання незавершених «битих» об'єктів.

## Кроки реалізації

1. Переконайтесь в тому, що створення різних відображень об'єкта можна звести до загальних кроків.
2. Опишіть ці кроки в загальному інтерфейсі будівельників.
3. Для кожного з відображень об'єкта-продукту створіть по одному класу-будівельнику й реалізуйте їхні методи будівництва.

Не забудьте про метод отримання результату. Зазвичай конкретні будівельники визначають власні методи отримання результату будівництва. Ви не можете описати ці методи в інтерфейсі будівельників, оскільки продукти не обов'язково повинні мати загальний базовий клас або інтерфейс. Але ви завжди можете додати метод отримання результату до загального інтерфейсу, якщо ваші будівельники виготовляють однорідні продукти, які мають спільного предка.

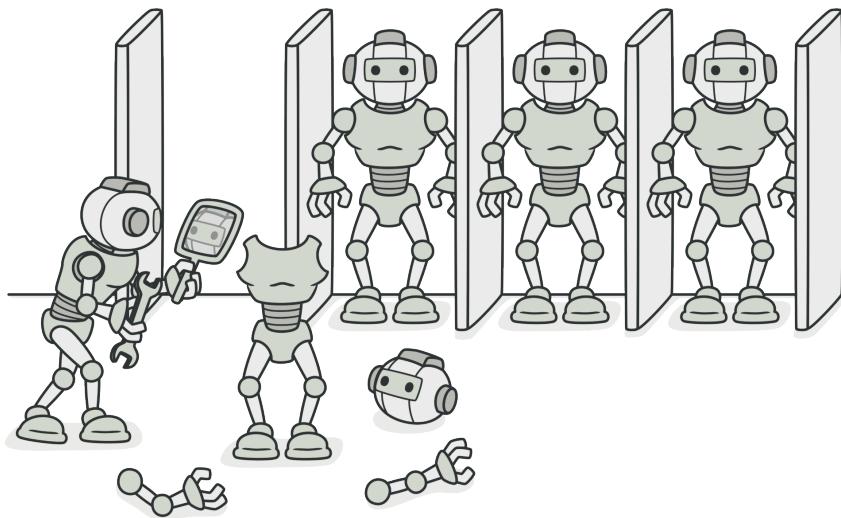
4. Подумайте про створення класу директора. Його методи створюватимуть різні конфігурації продуктів, викликаючи різні кроки одного і того самого будівельника.
5. Клієнтський код повинен буде створювати й об'єкти будівельників, й об'єкт директора. Перед початком будівництва клієнт повинен зв'язати певного будівельника з директором. Це можна зробити або через конструктор, або через сетер, або подавши будівельника безпосередньо до будівельного методу директора.
6. Результат будівництва можна повернути з директора, але тільки якщо метод повернення продукту вдалося розмістити в загальному інтерфейсі будівельників. Інакше ви жорстко прив'яжете директора до конкретних класів будівельників.

## **Δelta Переваги та недоліки**

- ✓ Дозволяє створювати продукти покроково.
- ✓ Дозволяє використовувати один і той самий код для створення різноманітних продуктів.
- ✓ Ізолює складний код конструювання продукту від його головної бізнес-логіки.
- ✗ Ускладнює код програми за рахунок додаткових класів.
- ✗ Клієнт буде прив'язаний до конкретних класів будівельників, тому що в інтерфейсі будівельника може не бути методу отримання результату.

## ↔ Відносини з іншими патернами

- Багато архітектур починаються із застосування Фабричного методу (простішого та більш розширеного за допомогою підкласів) та еволюціонують у бік Абстрактної фабрики, Прототипу або Будівельника (гнучкіших, але й складніших).
- Будівельник концентрується на будівництві складних об'єктів крок за кроком. Абстрактна фабрика спеціалізується на створенні сімейств пов'язаних продуктів. Будівельник повертає продукт тільки після виконання всіх кроків, а Абстрактна фабрика повертає продукт одразу.
- Будівельник дозволяє покроково конструювати дерево Компонувальника.
- Патерн Будівельник може бути побудований у вигляді Мосту: директор гриміте роль абстракції, а будівельники – реалізації.
- Абстрактна фабрика, Будівельник та Прототип можуть реалізовуватися за допомогою Одинака.



# ПРОТОТИП

Також відомий як: Клон, Prototype

**Прототип** – це породжувальний патерн проектування, що дає змогу копіювати об'єкти, не вдаючись у подробиці їхньої реалізації.

## (:() Проблема

У вас є об'єкт, який потрібно скопіювати. Як це зробити? Потрібно створити порожній об'єкт того самого класу, а потім по черзі копіювати значення всіх полів зі старого об'єкта до нового.

Чудово! Проте є нюанс. Не кожен об'єкт вдається скопіювати у такий спосіб, адже частина його стану може бути приватною, а значить – недоступною для решти коду програми.



*Копіювання «ззовні» не завжди можливе на практиці.*

Є їнша проблема. Код, що копіює, стане залежним від класів об'єктів, які він копіює. Адже, щоб перебрати усі поля об'єкта, потрібно прив'язатися до його класу. Тому ви не зможете копіювати об'єкти, знаючи тільки їхні інтерфейси, але не їхні конкретні класи.

## 😊 Рішення

Патерн Прототип доручає процес копіювання самим об'єктам, які треба скопіювати. Він вводить загальний інтерфейс для всіх об'єктів, що підтримують клонування. Це дозволяє копіювати об'єкти, не прив'язуючись до їхніх конкретних класів. Зазвичай такий інтерфейс має всього один метод – `clone`.

Реалізація цього методу в різних класах дуже схожа. Метод створює новий об'єкт поточного класу й копіює в нього значення всіх полів власного об'єкта. Таким чином можна скопіювати навіть приватні поля, оскільки більшість мов програмування дозволяє отримати доступ до приватних полів будь-якого об'єкта поточного класу.



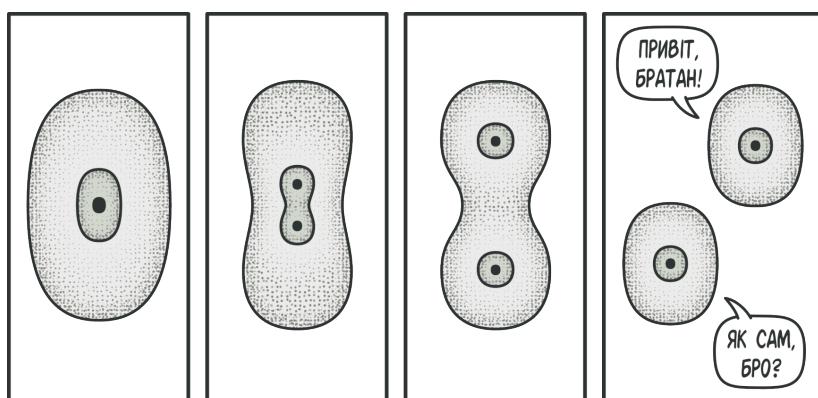
*Попередньо заготовлені прототипи можуть стати заміною підкласів.*

Об'єкт, який копіюють, називається *прототипом* (звідси і назва патерна). Коли об'єкти програми містять сотні полів і тисячі можливих конфігурацій, прототипи можуть слугувати своєрідною альтернативою створенню підкласів.

У цьому випадку всі можливі прототипи готуються і налаштовуються на етап ініціалізації програми. Потім, коли програмі буде потрібний новий об'єкт, вона створить копію з попередньо заготовленого прототипа.

## Аналогія з життя

У промисловому виробництві прототипи створюються перед виготовленням основної партії продуктів для проведення різноманітних випробувань. При цьому прототип не бере участі в подальшому виробництві, відіграючи пасивну роль.

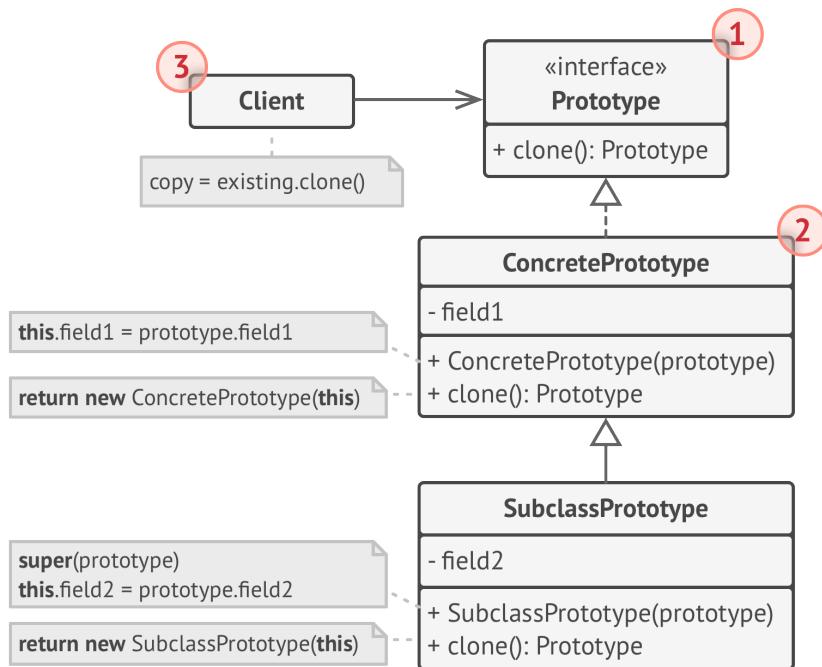


Приклад поділу клітини.

Виробничий прототип не створює копію самого себе, тому більш наближений до патерна приклад – це поділ клітин. Після мітозного поділу клітин утворюються дві абсолютно ідентичні клітини. Материнська клітина відіграє роль прототипу, беручи активну участь у створенні нового об'єкта.

## Структура

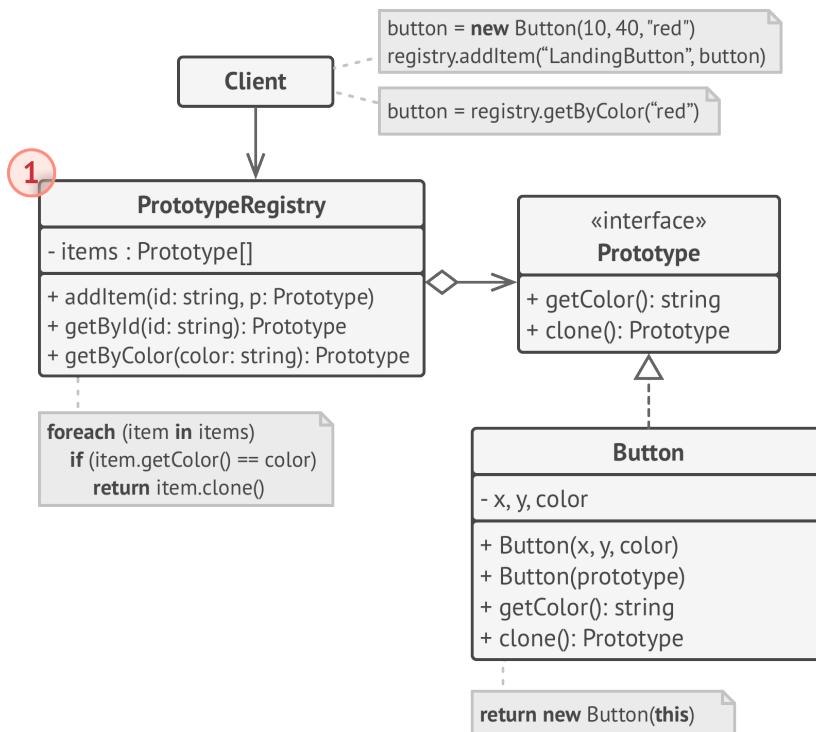
### Базова реалізація



- Інтерфейс **прототипів** описує операції клонування. Для більшості випадків – це єдиний метод `clone`.

2. **Конкретний прототип** реалізує операцію клонування самого себе. Крім звичайного копіювання значень усіх полів, тут можуть бути приховані різноманітні складнощі, про які клієнту не потрібно знати. Наприклад, клонування пов'язаних об'єктів, розплутування рекурсивних залежностей та інше.
3. **Клієнт** створює копію об'єкта, звертаючись до нього через загальний інтерфейс прототипів.

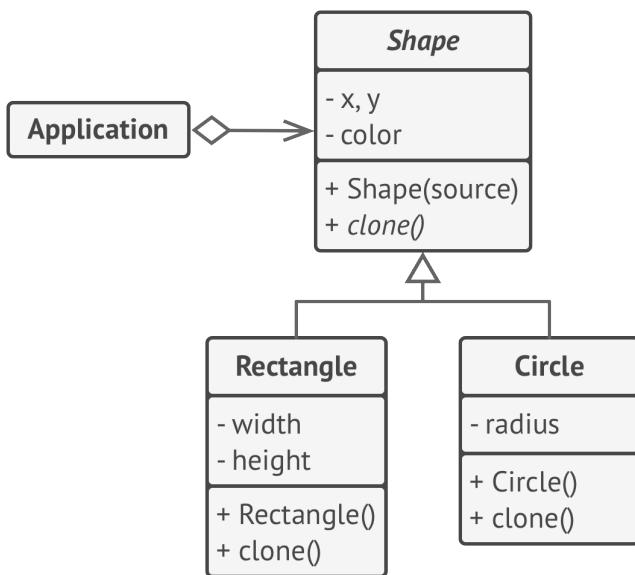
### Реалізація зі спільним сховищем прототипів



1. **Сховище прототипів** полегшує доступ до часто використовуваних прототипів, зберігаючи попередньо створений набір еталонних, готових до копіювання об'єктів. Найпростіше сховище може бути побудовано за допомогою хеш-таблиці виду `ім'я-прототипу → прототип`. Для полегшення пошуку прототипи можна маркувати ще й за іншими критеріями, а не тільки за умовним іменем.

## # Псевдокод

У цьому прикладі **Прототип** дозволяє робити точні копії об'єктів геометричних фігур без прив'язки до їхніх класів.



*Приклад клонування ієархії геометричних фігур.*

Кожна фігура реалізує інтерфейс клонування і надає метод для відтворення самої себе. Підкласи використовують

батьківський метод клонування, а потім копіюють власні поля до створеного об'єкта.

```
1 // Базовий прототип.  
2 abstract class Shape is  
3     field X: int  
4     field Y: int  
5     field color: string  
6  
7 // Звичайний конструктор.  
8 constructor Shape() is  
9     // ...  
10  
11 // Конструктор прототипа.  
12 constructor Shape(source: Shape) is  
13     this()  
14     this.X = source.X  
15     this.Y = source.Y  
16     this.color = source.color  
17  
18 // Результатом операції клонування завжди буде об'єкт з  
19 // ієрархії класів Shape.  
20 abstract method clone(): Shape  
21  
22  
23 // Конкретний прототип. Метод клонування створює новий об'єкт  
24 // поточного класу, передаючи до конструктора посилання на  
25 // власний об'єкт. Завдяки цьому, клонування виходить  
26 // атомарним – доки не виконається конструктор, нового об'єкта  
27 // ще не існує. Але як тільки конструктор завершено, ми  
28 // отримаємо завершений об'єкт-клон, а не порожній об'єкт, який
```

```
29 // потрібно ще заповнити.
30 class Rectangle extends Shape is
31     field width: int
32     field height: int
33
34 constructor Rectangle(source: Rectangle) is
35     // Виклик батьківського конструктора потрібен, щоб
36     // скопіювати потенційні приватні поля, оголошені в
37     // батьківському класі.
38     super(source)
39     this.width = source.width
40     this.height = source.height
41
42 method clone(): Shape is
43     return new Rectangle(this)
44
45
46 class Circle extends Shape is
47     field radius: int
48
49 constructor Circle(source: Circle) is
50     super(source)
51     this.radius = source.radius
52
53 method clone(): Shape is
54     return new Circle(this)
55
56
57 // Десь у клієнтському програмному коді.
58 class Application is
59     field shapes: array of Shape
60
```

```

61  constructor Application() is
62      Circle circle = new Circle()
63      circle.X = 10
64      circle.Y = 10
65      circle.radius = 20
66      shapes.add(circle)
67
68      Circle anotherCircle = circle.clone()
69      shapes.add(anotherCircle)
70      // anotherCircle буде містити точну копію circle.
71
72      Rectangle rectangle = new Rectangle()
73      rectangle.width = 10
74      rectangle.height = 20
75      shapes.add(rectangle)
76
77  method businessLogic() is
78      // Неочевидний плюс Прототипу в тому, що ви можете
79      // клонувати набір об'єктів, не знаючи їхніх конкретних
80      // класів.
81      Array shapesCopy = new Array of Shapes.
82
83      // Наприклад, ми не знаємо, які конкретно об'єкти
84      // знаходяться всередині масиву shapes так як його
85      // оголошено з типом Shape. Але завдяки поліморфізму, ми
86      // можемо клонувати усі об'єкти «наосліп». Буде виконано
87      // метод clone того класу, яким є цей об'єкт.
88      foreach (s in shapes) do
89          shapesCopy.add(s.clone())
90
91      // Змінна shapesCopy буде містити точні копії елементів
92      // масиву shapes.

```

## Застосування

 **Коли ваш код не повинен залежати від класів об'єктів, призначених для копіювання.**

 Таке часто буває, якщо ваш код працює з об'єктами, поданими ззовні через який-небудь загальний інтерфейс. Ви не зможете прив'язатися до їхніх класів, навіть якби захотіли, тому що конкретні класи об'єктів невідомі.

Патерн Прототип надає клієнту загальний інтерфейс для роботи з усіма прототипами. Клієнту не потрібно залежати від усіх класів об'єктів, призначених для копіювання, а тільки від інтерфейсу клонування.

 **Коли ви маєте безліч підкласів, які відрізняються початковими значеннями полів. Хтось міг створити усі ці класи для того, щоб мати легкий спосіб породжувати об'єкти певної конфігурації.**

 Патерн Прототип пропонує використовувати набір прототипів замість створення підкласів для опису популярних конфігурацій об'єктів.

Таким чином, замість породження об'єктів з підкласів ви копіюватимете існуючі об'єкти-прототипи, внутрішній стан яких вже налаштовано. Це дозволить уникнути вибухоподібного зростання кількості класів програми й зменшити її складність.

## Кроки реалізації

1. Створіть інтерфейс прототипів з єдиним методом `clone`. Якщо у вас вже є ієрархія продуктів, метод клонування можна оголосити в кожному з її класів.
2. Додайте до класів майбутніх прототипів альтернативний конструктор, що приймає в якості аргументу об'єкт поточного класу. Спочатку цей конструктор повинен скопіювати значення всіх полів поданого об'єкта, оголошених в рамках поточного класу. Потім – передати виконання батьківському конструктору, щоб той потурбувався про поля, оголошені в супер класі.

Якщо мова програмування, яку ви використовуєте, не підтримує перевантаження методів, тоді вам не вдасться створити декілька версій конструктора. В цьому випадку копіювання значень можна проводити в іншому методі, спеціально створеному для цих цілей. Конструктор є зручнішим, тому що дозволяє клонувати об'єкт за один виклик.

3. Зазвичай метод клонування складається з одного рядка, а саме виклику оператора `new` з конструктором прототипу. Усі класи, що підтримують клонування, повинні явно визначити метод `clone` для того, щоб вказати власний клас з оператором `new`. Інакше результатом клонування стане об'єкт батьківського класу.

4. На додачу можете створити центральне сховище прототипів. У ньому зручно зберігати варіації об'єктів, можливо, навіть одного класу, але по-різному налаштованих.

Ви можете розмістити це сховище або у новому фабричному класі, або у фабричному методі базового класу прототипів. Такий фабричний метод, керуючись вхідними аргументами, повинен шукати відповідний екземпляр у сховищі прототипів, а потім викликати його метод клонування і повернати отриманий об'єкт.

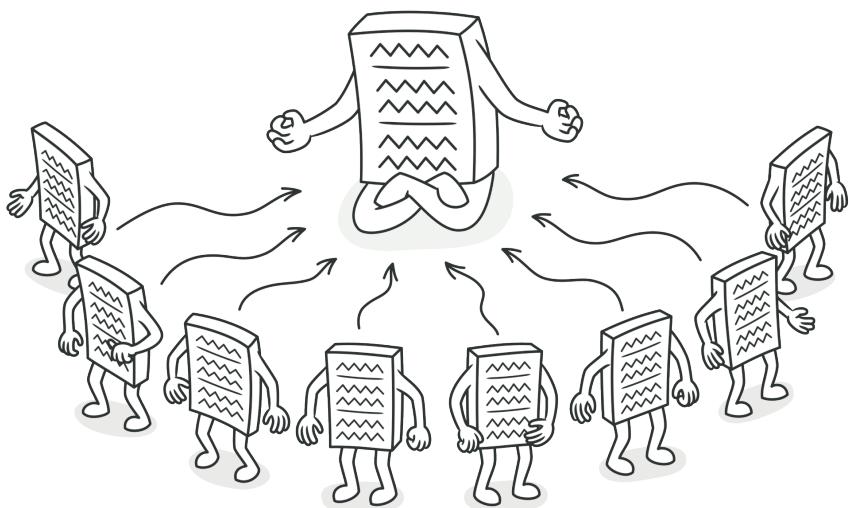
Нарешті, потрібно позбутися прямих викликів конструкторів об'єктів, замінивши їх викликами фабричного методу сховища прототипів.

## Переваги та недоліки

- ✓ Дозволяє клонувати об'єкти без прив'язки до їхніх конкретних класів.
- ✓ Менша кількість повторювань коду ініціалізації об'єктів.
- ✓ Прискорює створення об'єктів.
- ✓ Альтернатива створенню підкласів під час конструювання складних об'єктів.
- ✗ Складно клонувати складові об'єкти, що мають посилання на інші об'єкти.

## ↔ Відносини з іншими патернами

- Багато архітектур починаються із застосування Фабричного методу (простішого та більш розширеного за допомогою підкласів) та еволюціонують у бік Абстрактної фабрики, Прототипу або Будівельника (гнучкіших, але й складніших).
- Класи Абстрактної фабрики найчастіше реалізуються за допомогою Фабричного методу, хоча вони можуть бути побудовані і на основі Прототипу.
- Якщо Команду потрібно копіювати перед вставкою в історію виконаних команд, вам може допомогти Прототип.
- Архітектура, побудована на Компонувальниках та Декораторах, часто може поліпшуватися за рахунок впровадження Прототипу. Він дозволяє клонувати складні структури об'єктів, а не збирати їх заново.
- Прототип не спирається на спадкування, але йому потрібна складна операція ініціалізації. Фабричний метод, навпаки, побудований на спадкуванні, але не вимагає складної ініціалізації.
- Знімок іноді можна замінити Прототипом, якщо об'єкт, чий стан потрібно зберігати в історії, досить простий, не має посилань на зовнішні ресурси або їх можна легко відновити.
- Абстрактна фабрика, Будівельник та Прототип можуть реалізовуватися за допомогою Однака.



# ОДИНАК

Також відомий як: *Singleton*

**Однак** – це породжувальний патерн проектування, який гарантує, що клас має лише один екземпляр, та надає глобальну точку доступу до нього.

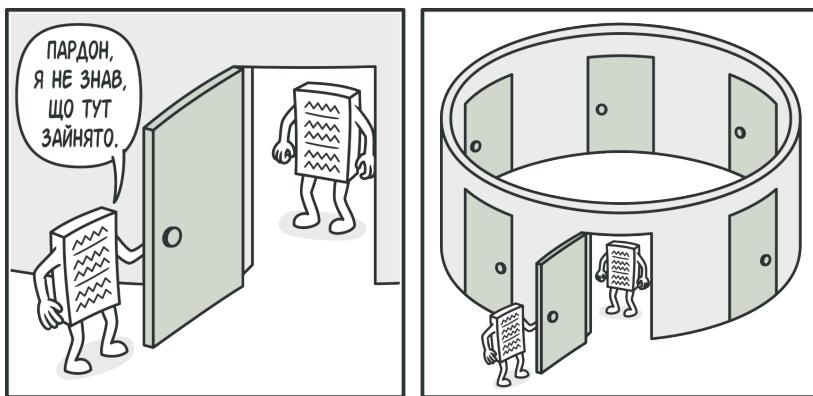
## (:() Проблема

Однак вирішує відразу дві проблеми (порушуючи *принцип єдиного обов'язку класу*):

1. **Гарантую наявність єдиного екземпляра класу.** Найчастіше за все це корисно для доступу до якогось спільногоресурсу, наприклад, бази даних.

Уявіть собі, що ви створили об'єкт, а через деякий час намагаєтесь створити ще один. У цьому випадку хотілося б отримати старий об'єкт замість створення нового.

Таку поведінку неможливо реалізувати за допомогою звичайного конструктора, оскільки конструктор класу **завжди** повертає новий об'єкт.



Клієнти можуть не підозрювати, що працюють з одним і тим самим об'єктом.

2. **Надає глобальну точку доступу.** Це не просто глобальна змінна, через яку можна дістатися до певного об'єкта. Глобальні змінні не захищені від запису, тому будь-який код може підмінити їхнє значення без вашого відома.

Проте, є ще одна особливість. Було б непогано й зберігати в одному місці код, який вирішує проблему №1, і мати до нього простий та доступний інтерфейс.

Цікаво, що в наш час патерн став настільки відомим, що тепер люди називають «одинаками» навіть ті класи, які вирішують лише одну з проблем, перерахованих вище.

## Рішення

Всі реалізації Одинака зводяться до того, аби приховати типовий конструктор та створити публічний статичний метод, який і контролюватиме життєвий цикл об'єкта-одинака.

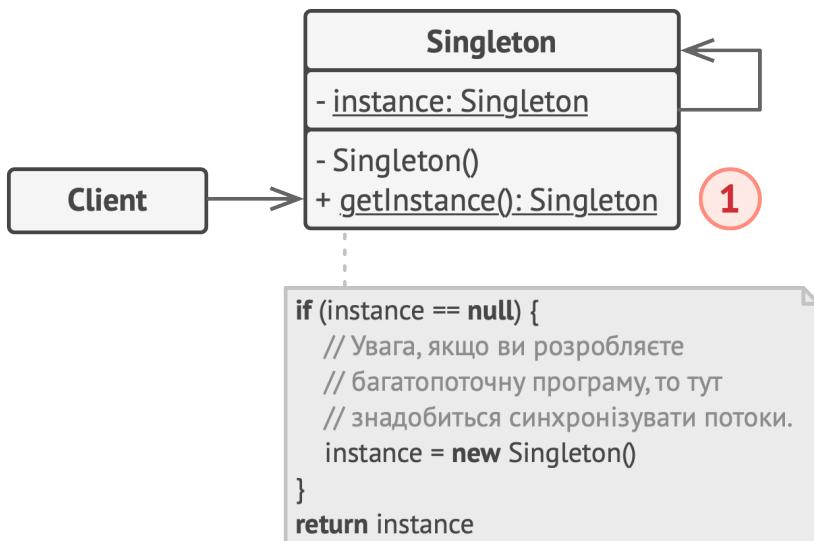
Якщо у вас є доступ до класу одинака, отже, буде й доступ до цього статичного методу. З якої точки коду ви б його не викликали, він завжди віддаватиме один і той самий об'єкт.

## Аналогія з життя

Уряд держави – вдалий приклад Одинака. У державі може бути тільки один офіційний уряд. Незалежно від того, хто

конкретно засідає в уряді, він має глобальну точку доступу «Уряд країни N».

## Структура



- Однак** визначає статичний метод `getInstance`, який повертав один екземпляр свого класу.

Конструктор Однака повинен бути прихований від клієнтів. Виклик методу `getInstance` повинен стати єдиним способом отримати об'єкт цього класу.

## # Псевдокод

У цьому прикладі роль **Однака** грає клас підключення до бази даних.

Цей клас не має публічного конструктора, тому єдиним способом отримання його об'єкта є виклик методу `getInstance`. Цей метод збереже перший створений об'єкт і повертає його в усіх наступних викликах.

```

1 // Клас одинака визначає статичний метод `getInstance`, котрий
2 // дозволяє клієнтам повторно використовувати одне і теж
3 // підключення до бази даних по всій програмі.
4 class Database is
5     // Поле для зберігання об'єкта–одинака має бути оголошено
6     // статичним.
7     private static field instance: Database
8
9     // Конструктор одинака завжди повинен залишатися приватним,
10    // аби клієнти не могли самостійно створювати екземпляри
11    // цього класу через оператор `new`.
12    private constructor Database() is
13        // Тут може жити код ініціалізації підключення до
14        // сервера баз даних.
15        // ...
16
17    // Головний статичний метод одинака служить альтернативою
18    // конструктору і є точкою доступу до екземпляра цього
19    // класу.
20    public static method getInstance() is
21        if (Database.instance == null) then
22            acquireThreadLock() and then
23                // Про всякий випадок, ще раз перевіримо, чи не
24                // було створено об'єкт в іншому потоці, поки
25                // даний потік чекав на звільнення блокування.
26        if (Database.instance == null) then
```

```

27         Database.instance = new Database()
28     return Database.instance
29
30     // І, нарешті, будь-який клас одинака повинен мати якусь
31     // корисну функціональність, яку клієнти будуть запускати
32     // через отриманий об'єкт одинака.
33     public method query(sql) is
34         // Усі запити до бази даних проходитимуть через цей
35         // метод. Тому є сенс помістити сюди якусь логіку
36         // кешування.
37         // ...
38
39     class Application is
40         method main() is
41             Database foo = Database.getInstance()
42             foo.query("SELECT ...")
43             // ...
44             Database bar = Database.getInstance()
45             bar.query("SELECT ...")
46             // Змінна "bar" містить той самий об'єкт, що і змінна
47             // "foo".

```

## 💡 Застосування

- ⚡ Коли в програмі повинен бути єдиний екземпляр якого-небудь класу, доступний усім клієнтам (наприклад, спільний доступ до бази даних з різних частин програми).
- ⚡ Однак приховує від клієнтів всі способи створення нового об'єкта, окрім спеціального методу. Цей метод або ство-

рює об'єкт, або віддає існуючий об'єкт, якщо він вже був створений.

### **Коли ви хочете мати більше контролю над глобальними змінними.**

 На відміну від глобальних змінних, Однак гарантує, що жоден інший код не замінить створений екземпляр класу, тому ви завжди впевнені в наявності лише одного об'єкта-одинака.

Тим не менше, будь-коли ви можете розширити це обмеження і дозволити будь-яку кількість об'єктів-одинаків, змінивши код в одному місці (метод `getInstance` ).

### **Кроки реалізації**

1. Додайте до класу приватне статичне поле, котре міститиме одиночний об'єкт.
2. Оголосіть статичний створюючий метод, що використовувається для отримання Одинака.
3. Додайте «лініву ініціалізацію» (створення об'єкта під час першого виклику методу) до створюючого методу одинака.
4. Зробіть конструктор класу приватним.

- У клієнтському коді замініть прямі виклики конструктора одинака на виклики його створюючого методу.

## ΔΔ Переваги та недоліки

- ✓ Гарантує наявність єдиного екземпляра класу.
- ✓ Надає глобальну точку доступу до нього.
- ✓ Реалізує відкладену ініціалізацію об'єкта-одинака.
- ✗ Порушує *принцип єдиного обов'язку класу*.
- ✗ Маскує поганий дизайн.
- ✗ Проблеми багатопоточності.
- ✗ Вимагає постійного створення Mock-об'єктів при юніт-тестуванні.

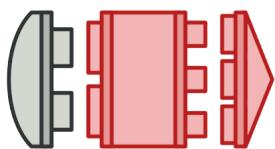
## ↔ Відносини з іншими патернами

- Фасад можна зробити **Однаком**, оскільки зазвичай потрібен тільки один об'єкт-фасад.
- Патерн **Легковаговик** може нагадувати **Однака**, якщо для конкретного завдання ви змогли зменшити кількість об'єктів до одного. Але пам'ятайте, що між патернами є дві суттєві відмінності:
  - На відміну від Однака, ви можете мати безліч об'єктів-легковаговиків.

2. Об'єкти-легковаговики повинні бути незмінними, тоді як об'єкт-одинаак допускає зміну свого стану.
- Абстрактна фабрика, Будівельник та Прототип можуть реалізовуватися за допомогою Одинаака.

# Структурні патерни

Ці патерни відповідають за побудову зручних в підтримці ієрархій класів.



## Адаптер

Adapter

Дає змогу об'єктам із несумісними інтерфейсами працювати разом.



## Міст

Bridge

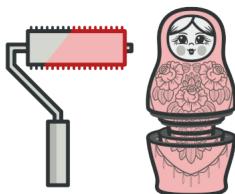
Розділяє один або кілька класів на дві окремі ієрархії – абстракцію та реалізацію, дозволяючи змінювати код в одній гілці класів, незалежно від іншої.



## Компонувальник

Composite

Дає змогу згрупувати декілька об'єктів у деревоподібну структуру, а потім працювати з нею так, ніби це одиничний об'єкт.



## Декоратор

Decorator

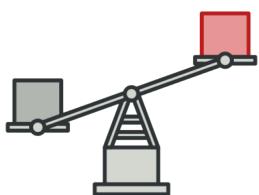
Дає змогу динамічно додавати об'єктам нову функціональність, загортуючи їх у корисні «обортки».



## Фасад

Facade

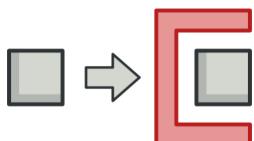
Надає простий інтерфейс до складної системи класів, бібліотеки або фреймворку.



## Легковаговик

Flyweight

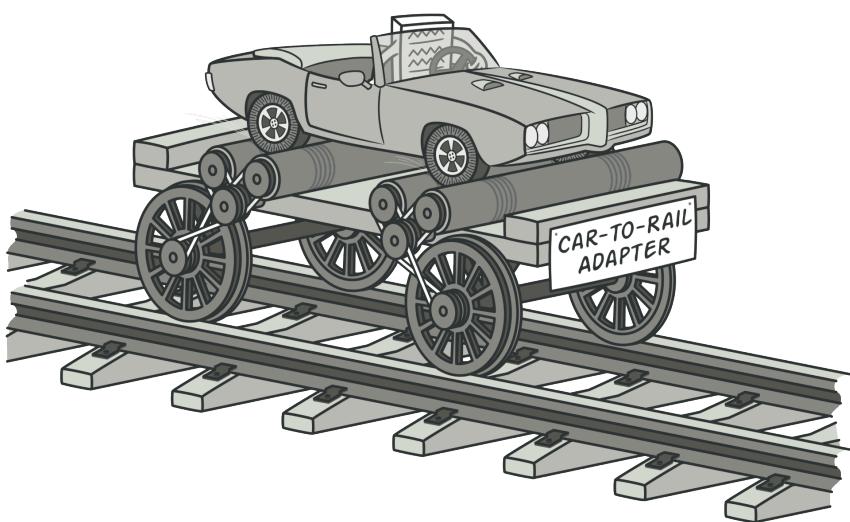
Дає змогу вмістити більшу кількість об'єктів у відведеній операційній пам'яті. Легковаговик заощаджує пам'ять, розподіляючи спільний стан об'єктів між собою, замість зберігання однакових даних у кожному об'єкті.



## Замісник

Proxy

Дає змогу підставляти замість реальних об'єктів спеціальні об'єкти-замінники. Ці об'єкти перехоплюють виклики до оригінального об'єкта, дозволяючи зробити щось до чи після передачі виклику оригіналові.



# АДАПТЕР

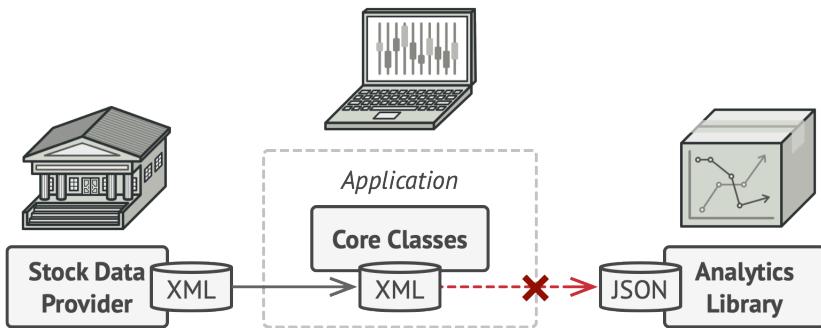
Також відомий як: *Wrapper, Обгортка, Adapter*

**Адаптер** – це структурний патерн проектування, що дає змогу об'єктам із несумісними інтерфейсами працювати разом.

## :( Проблема

Уявіть, що ви пишете програму для торгівлі на біржі. Ваша програма спочатку завантажує біржові котирування з декількох джерел в XML, а потім малює гарні графіки.

У якийсь момент ви вирішуєте покращити програму, застосувавши сторонню бібліотеку аналітики. Але от біда – бібліотека підтримує тільки формат даних JSON, несумісний із вашим додатком.



*Під'єднати сторонню бібліотеку неможливо через несумісність форматів даних.*

Ви могли б переписати цю бібліотеку, щоб вона підтримувала формат XML, але, по-перше, це може порушити роботу наявного коду, який уже залежить від бібліотеки, по-друге, у вас може просто не бути доступу до її вихідного коду.

## 😊 Рішення

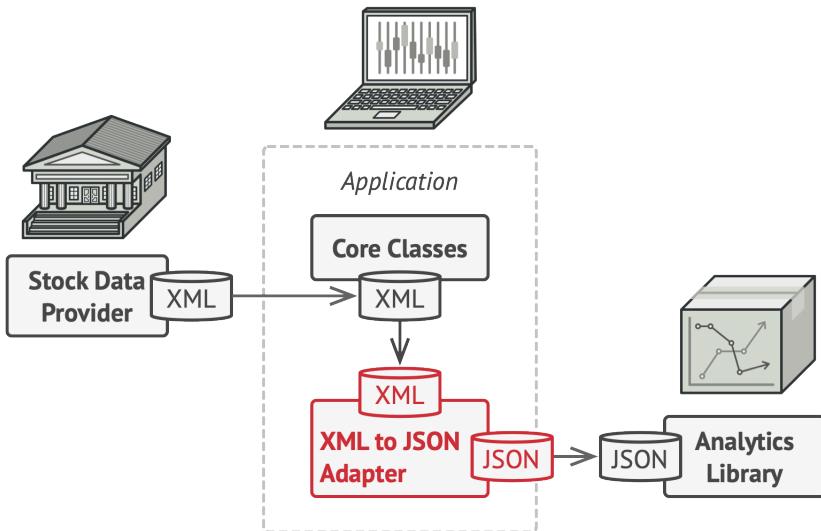
Ви можете створити *адаптер*. Це об'єкт-перекладач, який трансформує інтерфейс або дані одного об'єкта таким чином, щоб він став зрозумілим іншому об'єкту.

Адаптер загортає один з об'єктів так, що інший об'єкт навіть не підозрює про існування першого. Наприклад, об'єкт, що працює в метричній системі вимірювання, можна «обгорнути» адаптером, який буде конвертувати дані у тури.

Адаптери можуть не тільки конвертувати дані з одного формату в іншій, але й допомагати об'єктам із різними інтерфейсами працювати разом. Це виглядає так:

1. Адаптер має інтерфейс, сумісний з одним із об'єктів.
2. Тому цей об'єкт може вільно викликати методи адаптера.
3. Адаптер отримує ці виклики та перенаправляє їх іншому об'єкту, але вже в тому форматі та послідовності, які є зrozумілими для цього об'єкта.

Іноді вдається створити навіть *двосторонній адаптер*, який може працювати в обох напрямках.



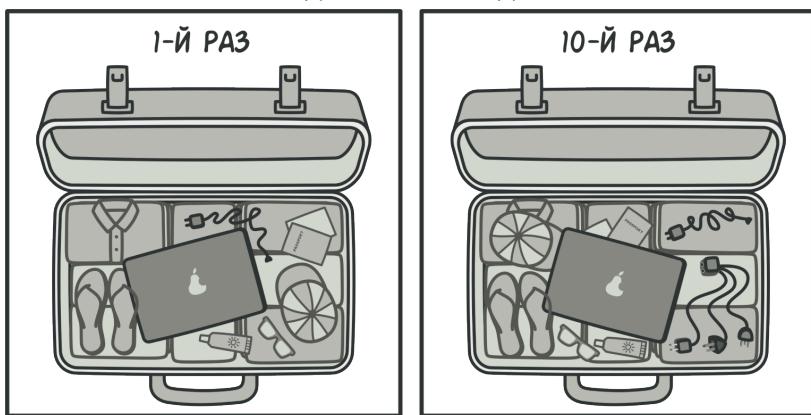
*Програма може працювати зі сторонньою бібліотекою через адаптер.*

Таким чином, для програми біржових котирувань ви могли б створити клас `XML_To_JSON_Adapter`, який би обгортав об'єкт того чи іншого класу бібліотеки аналітики. Ваш код посилав би адаптеру запити у форматі XML, а адаптер спочатку б транслював вхідні дані у формат JSON, а потім передавав їх методам загорнутого об'єкта аналітики.

## 🚗 Аналогія з життя

Під час вашої першої подорожі за кордон спроба зарядити ноутбук може стати неприємним сюрпризом, тому що стандарти розеток у багатьох країнах різняться.

## ПОДОРОЖ ЗА КОРДОН



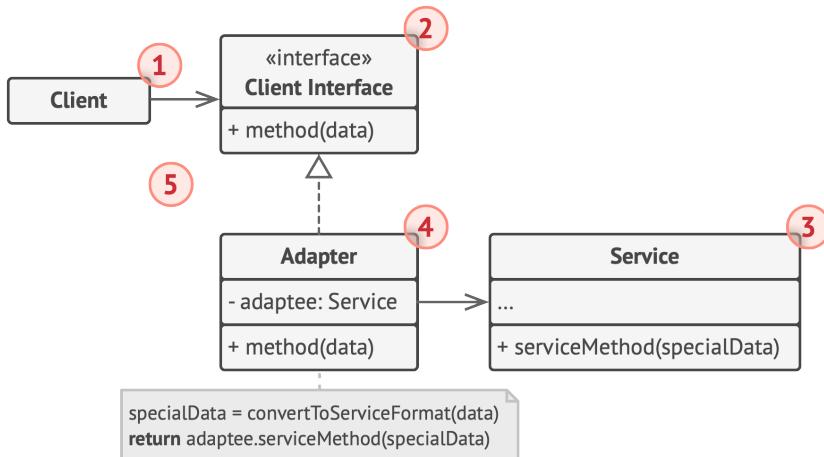
*Вміст валіз до й після поїздки за кордон.*

Ваша європейська зарядка стане непотрібом у США без спеціального адаптера, що дозволяє під'єднуватися до розетки іншого типу.

## STRUCTURE

### Адаптер об'єктів

Ця реалізація використовує агрегацію: об'єкт адаптера «загортався», тобто містить посилання на службовий об'єкт. Такий підхід працює в усіх мовах програмування.

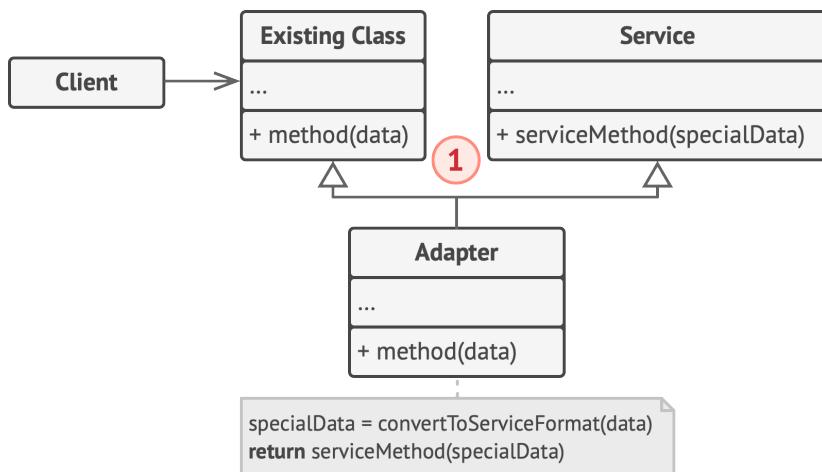


1. **Клієнт** – це клас, який містить існуючу бізнес-логіку програми.
2. **Клієнтський інтерфейс** описує протокол, через який клієнт може працювати з іншими класами.
3. **Сервіс** – це який-небудь корисний клас, зазвичай сторонній. Клієнт не може використовувати цей клас безпосередньо, оскільки сервіс має незрозумілий йому інтерфейс.
4. **Адаптер** – це клас, який може одночасно працювати і з клієнтом, і з сервісом. Він реалізує клієнтський інтерфейс і містить посилання на об'єкт сервісу. Адаптер отримує виклики від клієнта через методи клієнтського інтерфейсу, а потім конвертує їх у виклики методів загорнутого об'єкта в потрібному форматі.

5. Працюючи з адаптером через інтерфейс, клієнт не прив'язується до конкретного класу адаптера. Завдяки цьому ви можете додавати до програми нові види адаптерів, незалежно від клієнтського коду. Це може стати в нагоді, якщо інтерфейс сервісу раптом зміниться, наприклад, після виходу нової версії сторонньої бібліотеки.

## Адаптер класів

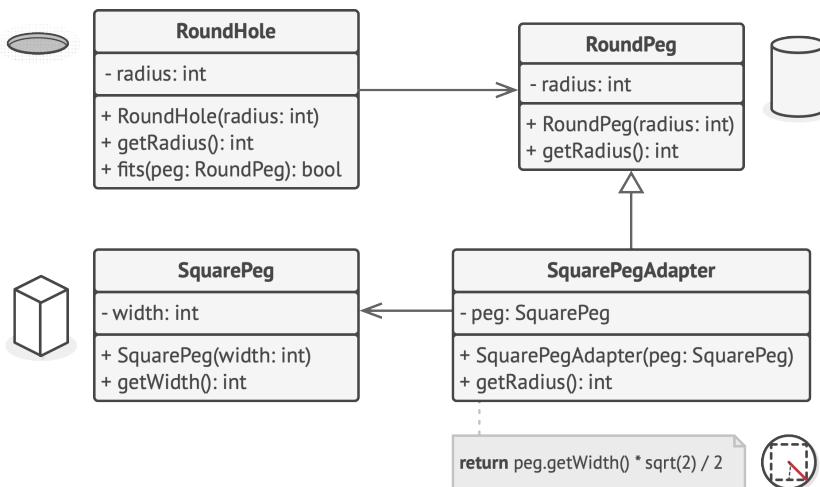
Ця реалізація базується на спадкуванні: адаптер успадковує обидва інтерфейси одночасно. Такий підхід можливий тільки в мовах, які підтримують множинне спадкування, наприклад у C++.



1. **Адаптер класів** не потребує вкладеного об'єкта, тому що він може одночасно успадковувати й частину існуючого класу, й частину класу сервісу.

## # Псевдокод

У цьому жартівливому прикладі **Адаптер** перетворює один інтерфейс на інший, дозволяючи поєднувати квадратні кілочки та круглі отвори.



*Приклад адаптації квадратних кілочків та круглих отворів.*

Адаптер обчислює найменший радіус кола, у яке можна вписати квадратний кілочок, і подає його як круглий кілочок із цим радіусом.

```

1 // Класи з сумісними інтерфейсами: КруглийОтвір та
2 // КруглийКілочок.
3 class RoundHole is
4   constructor RoundHole(radius) { ... }
5   method getRadius() is
6     // Повернути радіус отвору.
  
```

```
7  method fits(peg: RoundPeg) is
8      return this.getRadius() >= peg.getRadius()
9
10 class RoundPeg is
11     constructor RoundPeg(radius) { ... }
12
13     method getRadius() is
14         // Повернути радіус круглого кілочка.
15
16
17 // Застарілий несумісний клас: КвадратнийКілочок.
18 class SquarePeg is
19     constructor SquarePeg(width) { ... }
20
21     method getWidth() is
22         // Повернути ширину квадратного кілочка.
23
24
25 // Адаптер дозволяє використовувати квадратні кілочки й круглі
26 // отвори разом.
27 class SquarePegAdapter extends RoundPeg is
28     private field peg: SquarePeg
29
30     constructor SquarePegAdapter(peg: SquarePeg) is
31         this.peg = peg
32
33     method getRadius() is
34         // Обчислити половину діагоналі квадратного кілочка за
35         // теоремою Піфагора.
36         return peg.getWidth() * Math.sqrt(2) / 2
37
38
```

```

39 // Десь у клієнтському програмному коді.
40 hole = new RoundHole(5)
41 rpeg = new RoundPeg(5)
42 hole.fits(rpeg) // TRUE
43
44 small_sqpeg = new SquarePeg(5)
45 large_sqpeg = new SquarePeg(10)
46 hole.fits(small_sqpeg) // Помилка компіляції, несумісні типи.
47
48 small_sqpeg_adapter = new SquarePegAdapter(small_sqpeg)
49 large_sqpeg_adapter = new SquarePegAdapter(large_sqpeg)
50 hole.fits(small_sqpeg_adapter) // TRUE
51 hole.fits(large_sqpeg_adapter) // FALSE

```

## Застосування

-  Якщо ви хочете використати сторонній клас, але його інтерфейс не відповідає решті кодів програми.
-  Адаптер дозволяє створити об'єкт-прокладку, який перетворюватиме виклики програми у формат, зрозумілий сторонньому класу.
-  Якщо вам потрібно використати декілька існуючих підкласів, але в них не вистачає якої-небудь спільної функціональності, а розширити супер клас ви не можете.

-  Ви могли б створити ще один рівень підкласів та додати до них забраклу функціональність. Але при цьому доведеться дублювати один і той самий код в обох гілках підкласів.

Більш елегантним рішенням було б розмістити відсутню функціональність в адаптері й пристосувати його для роботи із супер класом. Такий адаптер зможе працювати з усіма підкласами ієархії. Це рішення сильно нагадуватиме патерн Декоратор.

## Кроки реалізації

1. Переконайтесь, що у вас є два класи з незручними інтерфейсами:
  - корисний *сервіс* – службовий клас, який ви не можете змінювати (він або сторонній, або від нього залежить інший код);
  - один або декілька *клієнтів* – існуючих класів програми, які не можуть використовувати сервіс через несумісний із ним інтерфейс.
2. Опишіть клієнтський інтерфейс, через який класи програм могли б використовувати клас сервісу.
3. Створіть клас адаптера, реалізувавши цей інтерфейс.
4. Розмістіть в адаптері поле, що міститиме посилання на об'єкт сервісу. Зазвичай це поле заповнюють об'єктом, переданим

у конструктор адаптера. Але цей об'єкт можна передавати й безпосередньо до методів адаптера.

5. Реалізуйте всі методи клієнтського інтерфейсу в адаптері. Адаптер повинен делегувати основну роботу сервісу.
6. Програма повинна використовувати адаптер тільки через клієнтський інтерфейс. Це дозволить легко змінювати та додавати адаптери в майбутньому.

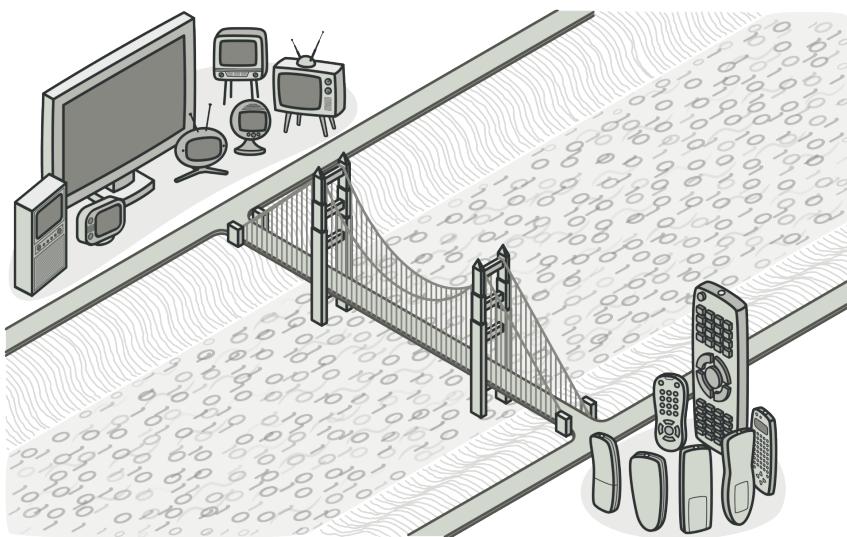
## **Переваги та недоліки**

- ✓ Відокремлює та приховує від клієнта подробиці перетворення різних інтерфейсів.
- ✗ Ускладнює код програми внаслідок введення додаткових класів.

## **Відносини з іншими патернами**

- **Міст** проектиють заздалегідь, щоб розвивати великі частини програми окремо одну від одної. **Адаптер** застосовується постфактум, щоб змусити несумісні класи працювати разом.
- **Адаптер** змінює інтерфейс існуючого об'єкта. **Декоратор** покращує інший об'єкт без зміни його інтерфейсу. Причому **Декоратор** підтримує рекурсивну вкладуваність, на відміну від **Адаптеру**.

- **Адаптер** надає класу альтернативний інтерфейс. **Декоратор** надає розширений інтерфейс. **Замісник** надає той самий інтерфейс.
- **Фасад** задає новий інтерфейс, тоді як **Адаптер** повторно використовує старий. **Адаптер** обгортає тільки один клас, а **Фасад** обгортає цілу підсистему. Крім того, **Адаптер** дозволяє двом існуючим інтерфейсам працювати спільно, замість того, щоб визначити повністю новий.
- **Міст**, **Стратегія** та **Стан** (а також трохи і **Адаптер**) мають схожі структури класів – усі вони побудовані за принципом «композиції», тобто делегування роботи іншим об'єктам. Проте вони відрізняються тим, що вирішують різні проблеми. Пам'ятайте, що патерни – це не тільки рецепт побудови коду певним чином, але й описування проблем, які призвели до такого рішення.



# MIST

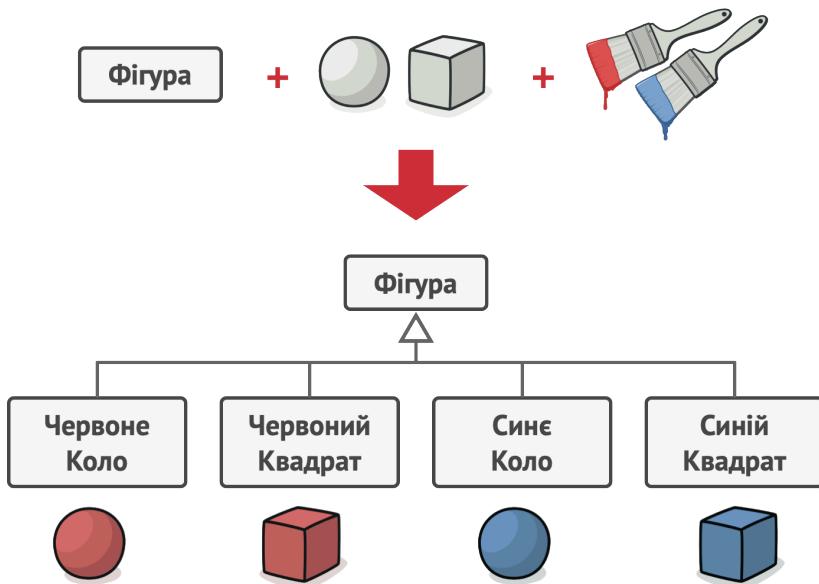
Також відомий як: *Bridge*

**Міст** – це структурний патерн проектування, який розділяє один або кілька класів на дві окремі ієархії – абстракцію та реалізацію, дозволяючи змінювати код в одній гілці класів, незалежно від іншої.

## :( Проблема

*Абстракція? Реалізація?! Звучить страхітливо! Розгляньмо простенький приклад, щоб зрозуміти про що йде мова.*

У вас є клас геометричних **Фігур**, який має підкласи **Круг** та **Квадрат**. Ви хочете розширити ієархію фігур за кольором, тобто мати **Червоні** та **Сині** фігури. Але для того, щоб все це об'єднати, доведеться створити 4 комбінації підкласів на зразок **СиніКруги** та **ЧервоніКвадрати**.



*Кількість підкласів зростає в геометричній прогресії.*

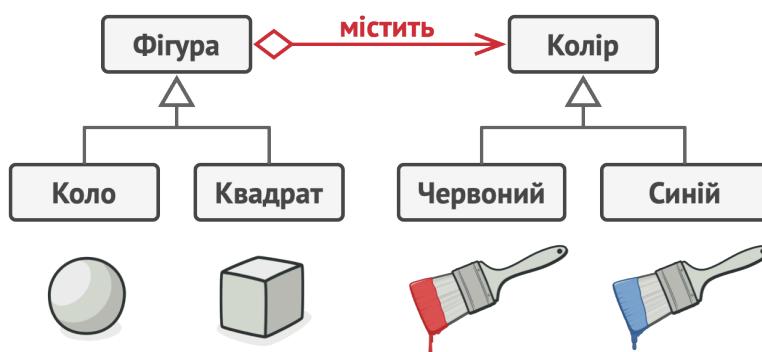
При додаванні нових видів фігур і кольорів кількість комбінацій зростатиме в геометричній прогресії. Наприклад, щоб ввести в програму фігури трикутників, доведеться створити

відразу два нових класи трикутників, по одному для кожного кольору. Після цього введення нового кольору вимагатиме створення вже трьох класів, по одному для кожного виду фігур. Чим далі, тим гірше.

## 😊 Рішення

Корінь проблеми полягає в тому, що ми намагаємося розширити класи фігур одразу в двох незалежних площинах – за видом та кольором. Саме це призводить до розростання дерева класів.

Патерн Міст пропонує замінити спадкування на делегування. Для цього потрібно виділити одну з таких «площин» в окрему ієархію і посилатися на об'єкт цієї ієархії, замість зберігання його стану та поведінки всередині одного класу.



*Розмноження підкласів можна зупинити, розбивши класи на кілька ієархій.*

Таким чином, ми можемо зробити `Колір` окремим класом з підкласами `Червоний` та `Синій`. Клас `Фігур` отримає посилання на об'єкт `Кольору` і зможе делегувати йому роботу, якщо виникне така необхідність. Такий зв'язок і стане мостом між `Фігурами` та `Кольором`. При додаванні нових класів кольорів не потрібно буде звертатись до класів фігур і навпаки.

## Абстракція і Реалізація

Ці терміни було введено в книзі GoF<sup>1</sup> при описі Мосту. На мій погляд, вони виглядають занадто академічними та показують патерн складнішим, ніж він є насправді. Пам'ятаючи про приклад з фігурами й кольорами, давайте все ж таки розберемося, що мали на увазі автори патерна.

Отже, *абстракція* (або *інтерфейс*) – це уявний рівень керування чим-небудь, що не виконує роботу самостійно, а делегує її рівню *реалізації* (який зветься *платформою*).

Тільки не плутайте ці терміни з *інтерфейсами* або *абстрактними класами* вашої мови програмування – це не одне і те ж саме.

---

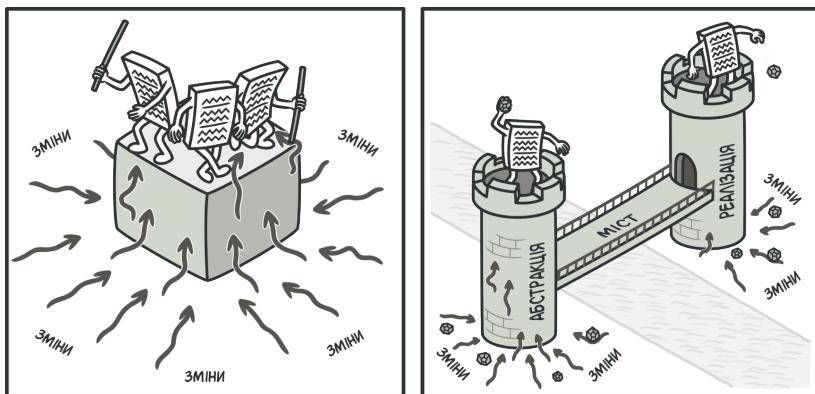
1. Gang of Four / «Банда четирьох». Автори книги *Design Patterns: Elements of Reusable Object-Oriented Software* <https://refactoring.guru/uk/gof-book>.

Якщо говорити про реальні програми, то абстракцією може виступати графічний інтерфейс програми (GUI), а реалізацією – низькорівневий код операційної системи (API), до якого графічний інтерфейс звертається, реагуючи на дії користувача.

Ви можете розвивати програму у двох різних напрямках:

- мати кілька різних GUI (наприклад, для звичайних користувачів та адміністраторів).
- підтримувати багато видів API (наприклад, працювати під Windows, Linux і macOS).

Така програма може виглядати як один великий клубок коду, в якому змішано умовні оператори рівнів GUI та API.

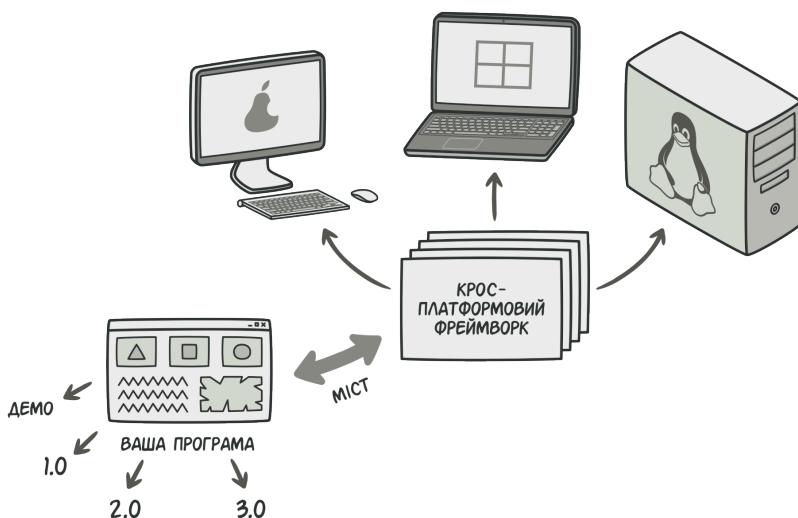


*Коли зміни беруть проект в «осаду», вам легше відбиватися, якщо розділити монолітний код на частини.*

Ви можете спробувати структурувати цей хаос, створивши для кожної з варіацій інтерфейсу-платформи свої підкласи. Але такий підхід призведе до зростання класів комбінацій, і з кожною новою платформою їх буде все більше й більше.

Ми можемо вирішити цю проблему, застосувавши Міст. Патерн пропонує розплутати цей код, розділивши його на дві частини:

- Абстракцію: рівень графічного інтерфейсу програми.
- Реалізацію: рівень взаємодії з операційною системою.

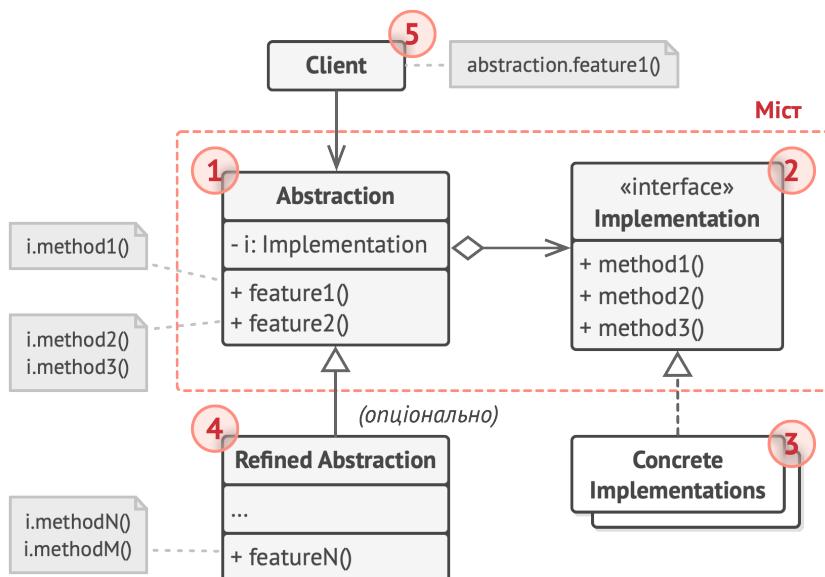


*Один з варіантів крос-платформової архітектури.*

Абстракція делегуватиме роботу одному з об'єктів реалізації. Причому, реалізації можна буде взаємозамінити, але тільки за умови, що всі вони слідуватимуть єдиному інтерфейсу.

Таким чином, ви зможете змінювати графічний інтерфейс програми, не чіпаючи низькорівневий код роботи з операційною системою. І навпаки, ви зможете додавати підтримку нових операційних систем, створюючи нові підкласи реалізації, без необхідності правити код у класах графічного інтерфейсу.

## Структура



1. **Абстракція** містить керуючу логіку. Код абстракції делегує реальну роботу пов'язаному об'єктові реалізації.
2. **Реалізація** описує загальний інтерфейс для всіх реалізацій. Всі методи, які тут описані, будуть доступні з класу абстракції та його підкласів.

Інтерфейси абстракції та реалізації можуть або збігатися, або бути абсолютно різними. Проте, зазвичай в реалізації живуть базові операції, на яких будуються складні операції абстракції.

3. **Конкретні реалізації** містять платформо-залежний код.
4. **Розширені абстракції** містять різні варіації керуючої логіки. Як і батьківский клас, працює з реалізаціями тільки через загальний інтерфейс реалізацій.
5. **Клієнт** працює тільки з об'єктами абстракції. Не рахуючи початкового зв'язування абстракції з однією із реалізацій, клієнтський код не має прямого доступу до об'єктів реалізації.

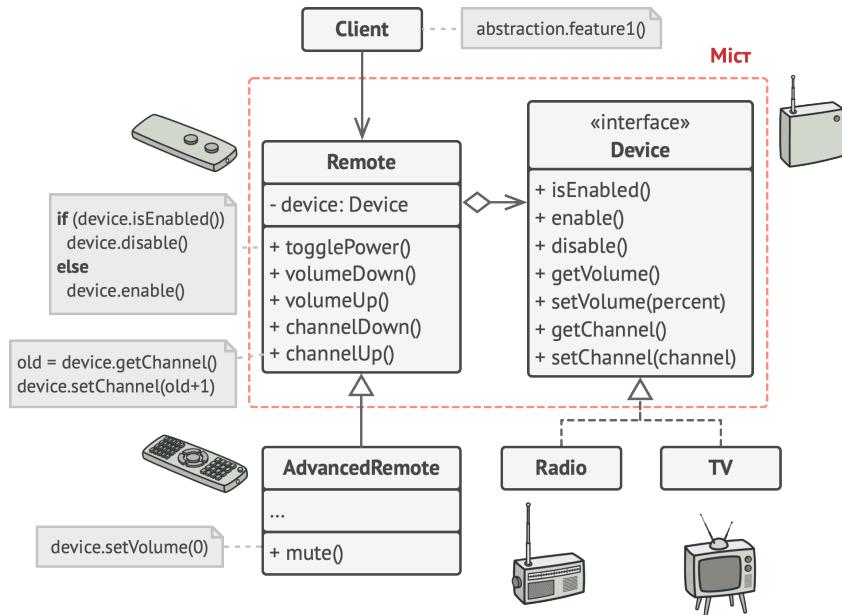
## # Псевдокод

У цьому прикладі **Міст** ділить монолітний код приладів та пультів на дві частини: прилади (виступають реалізацією) і пульти керування ними (виступають абстракцією).

Клас пульта має посилання на об'єкт приладу, яким він керує. Пульти працюють з приладами через загальний інтерфейс. Це дає можливість зв'язати пульти з різними приладами.

Пульти можна розвивати незалежно від приладів. Для цього достатньо створити новий підклас абстракції. Ви можете

створити як простий пульт з двома кнопками, так і більш складний пульт з тач-інтерфейсом.



Приклад поділу двох ієрархій класів – приладів та пультів керування.

Клієнтському коду залишається вибрати версію абстракції та реалізації, з якими він хоче працювати, та зв'язати їх між собою.

```

1 // Клас пультів має посилання на пристрій, яким керує. Методи
2 // цього класу делегують роботу методам пов'язаного пристроя.
3 class Remote is
4     protected field device: Device
5     constructor Remote(device: Device) is
6         this.device = device
  
```

```
7  method togglePower() is
8      if (device.isEnabled()) then
9          device.disable()
10     else
11         device.enable()
12 method volumeDown() is
13     device.setVolume(device.getVolume() - 10)
14 method volumeUp() is
15     device.setVolume(device.getVolume() + 10)
16 method channelDown() is
17     device.setChannel(device.getChannel() - 1)
18 method channelUp() is
19     device.setChannel(device.getChannel() + 1)
20
21
22 // Ви можете розширювати клас пультів, не чіпаючи код пристройів.
23 class AdvancedRemote extends Remote is
24     method mute() is
25         device.setVolume(0)
26
27
28 // Всі пристрої мають спільний інтерфейс, тому з ними може
29 // працювати будь-який пульт.
30 interface Device is
31     method isEnabled()
32     method enable()
33     method disable()
34     method getVolume()
35     method setVolume(percent)
36     method getChannel()
37     method setChannel(channel)
38
```

```

39 // Разом з цим, кожен пристрій має особливу реалізацію.
40 class Tv implements Device is
41     // ...
42
43 class Radio implements Device is
44     // ...
45
46
47 // Десять у клієнтському програмному коді.
48 tv = new Tv()
49 remote = new Remote(tv)
50 remote.togglePower()
51
52 radio = new Radio()
53 remote = new AdvancedRemote(radio)

```

## Застосування

-  Якщо ви хочете розділити монолітний клас, який містить кілька різних реалізацій якої-небудь функціональності (наприклад, якщо клас може працювати з різними системами баз даних).
-  Чим більший клас, тим важче розібратись у його коді, і тим більше це розтягує час розробки. Крім того, зміни, що вносяться в одну з реалізацій, призводять до редагування всього класу, що може викликати появу несподіваних помилок у коді.

Міст дозволяє розділити монолітний клас на кілька окремих ієрархій. Після цього ви можете змінювати код в одній гілці класів незалежно від іншої. Це спрощує роботу над кодом і зменшує ймовірність внесення помилок.

 **Якщо клас потрібно розширювати в двох незалежних площинах.**

 Міст пропонує виділити одну з таких площин в окрему ієрархію класів, зберігаючи посилання на один з її об'єктів у початковому класі.

 **Якщо ви хочете мати можливість змінювати реалізацію під час виконання програми.**

 Міст дозволяє замінювати реалізацію навіть під час виконання програми, оскільки конкретна реалізація не «зашита» в клас абстракції.

До речі, через цей пункт *Міст* часто плутають із Стратегією. Зверніть увагу, що у *Моста* цей пункт займає останнє місце за значущістю, оскільки його головна задача – структурна.

## Кроки реалізації

1. Визначте, чи існують у ваших класах два непересічних виміри. Це може бути функціональність/платформа, предметна

область/інфраструктура, фронт-енд/бек-енд або інтерфейс/реалізація.

2. Продумайте, які операції будуть потрібні клієнтам, і описіть їх у базовому класі *абстракції*.
3. Визначте поведінки, які доступні на всіх платформах, та виберіть з них ту частину, яка буде потрібна для абстракції. На підставі цього описіть загальний інтерфейс *реалізації*.
4. Дляожної платформи створіть власний клас конкретної реалізації. Всі вони повинні дотримуватися загального інтерфейсу, який ми виділили перед цим.
5. Додайте до класу абстракції посилання на об'єкт реалізації. Реалізуйте методи абстракції, делегуючи основну роботу пов'язаному об'єкту реалізації.
6. Якщо у вас є кілька варіацій абстракції, створіть дляожної з них власний підклас.
7. Клієнт повинен подати об'єкт реалізації до конструктора абстракції, щоб зв'язати їх разом. Після цього він може вільно використовувати об'єкт абстракції, забувши про реалізацію.

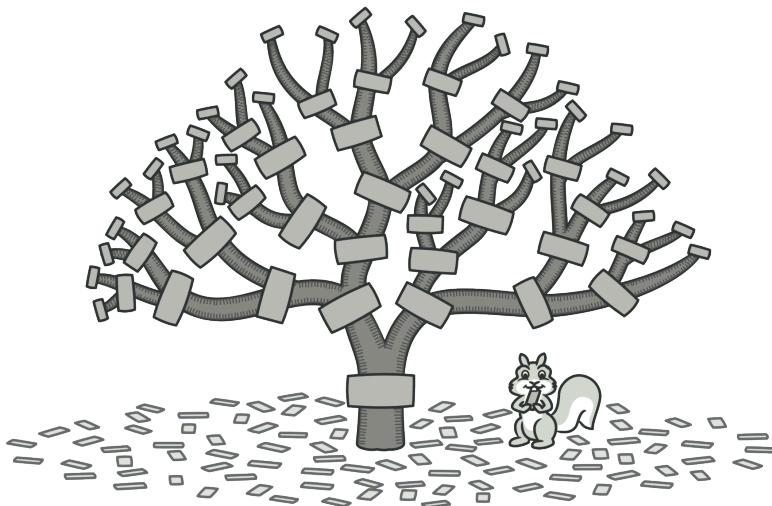
## **Переваги та недоліки**

- ✓ Дозволяє будувати платформо-незалежні програми.
- ✓ Приховує зайнві або небезпечні деталі реалізації від клієнтського коду.

- ✓ Реалізує *принцип відкритості/закритості*.
- ✗ Ускладнює код програми внаслідок введення додаткових класів.

## ↔ Відносини з іншими патернами

- **Міст** проектирують заздалегідь, щоб розвивати великі частини програми окремо одну від одної. **Адаптер** застосовується постфактум, щоб змусити несумісні класи працювати разом.
- **Міст, Стратегія та Стан** (а також трохи і **Адаптер**) мають схожі структури класів – усі вони побудовані за принципом «композиції», тобто делегування роботи іншим об'єктам. Проте вони відрізняються тим, що вирішують різні проблеми. Пам'ятайте, що патерни – це не тільки рецепт побудови коду певним чином, але й описування проблем, які призвели до такого рішення.
- **Абстрактна фабрика** може працювати спільно з **Мостом**. Це особливо корисно, якщо у вас є абстракції, які можуть працювати тільки з деякими реалізаціями. В цьому випадку фабрика визначатиме типи створюваних абстракцій та реалізацій.
- Патерн **Будівельник** може бути побудований у вигляді **Мосту**: директор гриміте роль абстракції, а будівельники – реалізації.



# КОМПОНУВАЛЬНИК

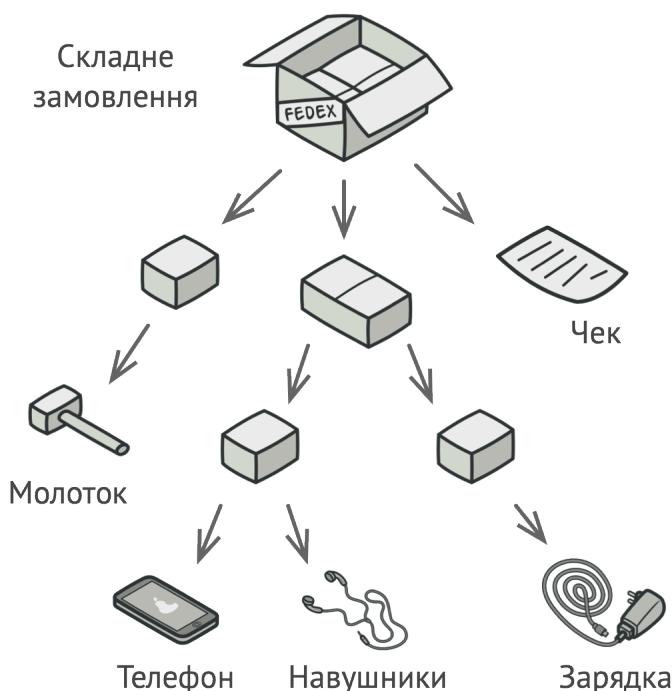
Також відомий як: *Дерево, Composite*

**Компонувальник** – це структурний патерн проектування, що дає змогу згрупувати декілька об'єктів у деревоподібну структуру, а потім працювати з нею так, ніби це одиничний об'єкт.

## :( Проблема

Патерн Компонувальник має сенс тільки в тих випадках, коли основна модель вашої програми може бути структурована у вигляді дерева.

Наприклад, є два об'єкти — **Продукт** і **Коробка**. **Коробка** може містити кілька **Продуктів** та інших **Коробок** меншого розміру. Останні, в свою чергу, також містять або **Продукти**, або **Коробки** і так далі.



Замовлення може складатися з різних продуктів, запакованих у власні коробки.

Тепер, припустімо, що ваші Продукти й Коробки можуть бути частиною замовлень. При цьому замовлення може містити як звичайні Продукт без пакування, так і наповнені змістом Коробки. Ваше завдання полягає в тому, щоб дізнатися вартість всього замовлення.

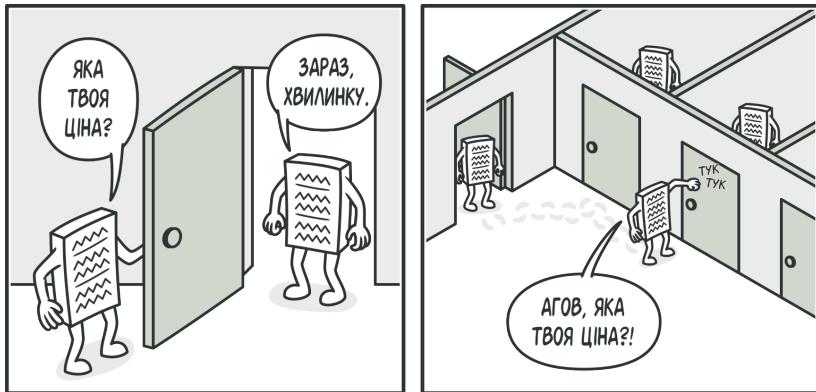
Якщо спробувати вирішити завдання напролом, тоді потрібно відкрити усі коробки замовлення, перебрати продукти й порахувати їхню загальну вартість. Але це занадто велика морока, оскільки типи коробок і їхній вміст можуть бути вам невідомі заздалегідь. Крім того, наперед невідомою є і кількість рівнів вкладеності коробок, тому перебрати коробки простим циклом не вийде.

## Рішення

Компонувальник пропонує розглядати Продукт і Коробку через єдиний інтерфейс зі спільним методом отримання ціни.

Продукт просто поверне свою вартість, а Коробка запише про вартість кожного предмета всередині себе і поверне суму результатів.

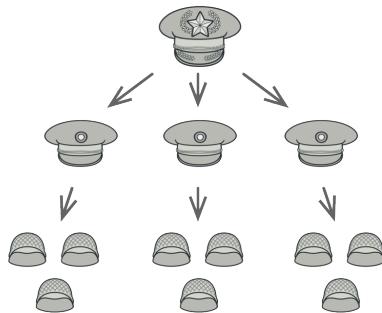
Якщо одним із внутрішніх предметів виявиться трохи менша коробка, вона теж буде перебирати власний вміст, і так далі, допоки не порахується вміст усіх складових частин.



Компонувальник рекурсивно запускає дію по всіх компонентах дерева – від коріння до листя.

Для вас як клієнта важливим є те, що вже не потрібно нічого знати про структуру замовлень. Ви викликаєте метод отримання ціни, він повертає цифру, і ви не «тонете» в горах картону та скотчу.

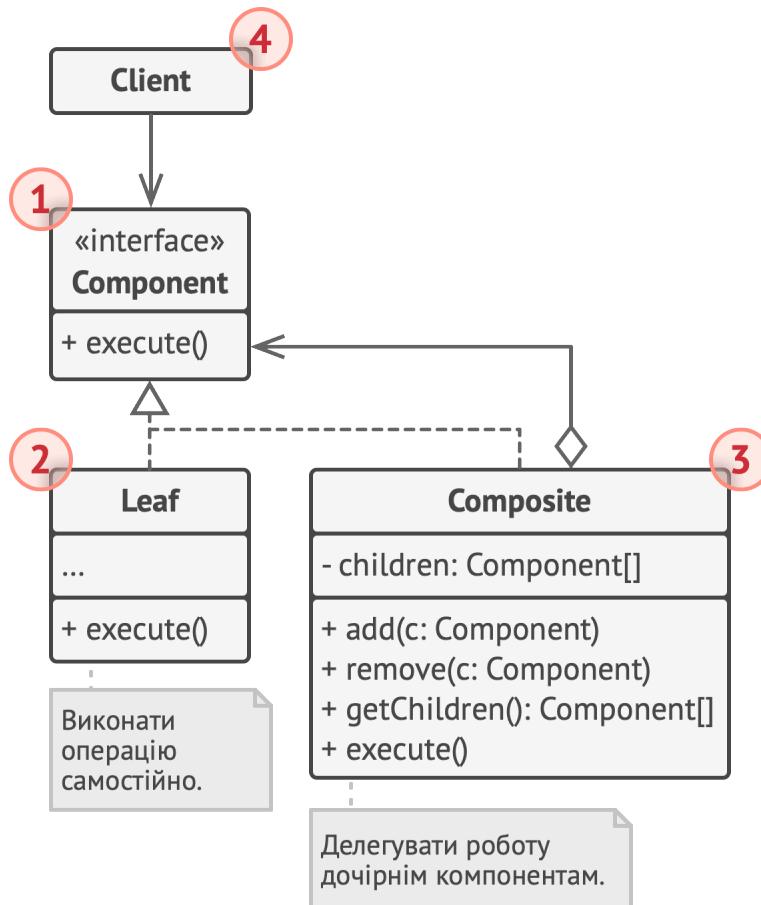
## 🚗 Аналогія з життя



Приклад армійської структури.

Армії більшості країн можуть бути представлені у вигляді перевернутих дерев. На нижньому рівні у вас солдати, далі взводи, далі полки, а далі цілі армії. Накази віддаються зверху вниз структурою командування до тих пір, поки вони не доходять до конкретного солдата.

## Структура



1. **Компонент** описує загальний інтерфейс для простих і складових компонентів дерева.
2. **Лист** – це простий компонент дерева, який не має відгальожень. Класи листя міститимуть більшу частину корисного коду, тому що їм нікому передавати його виконання.
3. **Контейнер** (або *композит*) – це складовий компонент дерева. Він містить набір дочірніх компонентів, але нічого не знає про їхні типи. Це можуть бути як прості компоненти-листя, так і інші компоненти-контейнери. Проте, це не проблема, якщо усі дочірні компоненти дотримуються єдиного інтерфейсу.

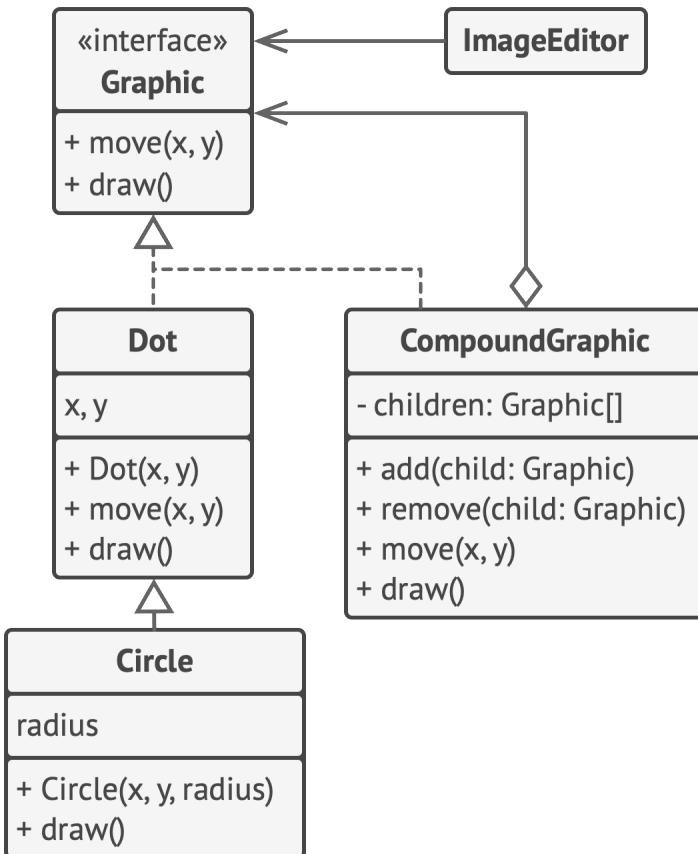
Методи контейнера переадресовують основну роботу своїм дочірнім компонентам, хоча можуть додавати щось своє до результату.

4. **Клієнт** працює з деревом через загальний інтерфейс компонентів.

Завдяки цьому, клієнту не важливо, що перед ним знаходиться – простий чи складовий компонент дерева.

## # Псевдокод

У цьому прикладі **Компонувальник** допомагає реалізувати вкладені геометричні фігури.



*Приклад редактора геометричних фігур.*

Клас **CompoundGraphic** може містити будь-яку кількість підфігур, включно з такими самими контейнерами, як і він сам. Контейнер реалізує ті ж самі методи, що і прості фігури. Але замість безпосередньої дії він передає виклики всім вкладеним компонентам, використовуючи рекурсію. Потім він як би «підсумовує» результати всіх вкладених фігур.

Клієнтський код працює з усіма фігурами через загальний інтерфейс фігур і не знає що перед ним – проста фігура

чи складова. Це дозволяє клієнтському коду працювати з деревами об'єктів будь-якої складності, не прив'язуючись до конкретних класів об'єктів, що формують дерево.

```

1 // Загальний інтерфейс компонентів.
2 interface Graphic is
3     method move(x, y)
4     method draw()
5
6 // Простий компонент.
7 class Dot implements Graphic is
8     field x, y
9
10 constructor Dot(x, y) { ... }
11
12 method move(x, y) is
13     this.x += x, this.y += y
14
15 method draw() is
16     // Намалювати крапку у координатах X, Y.
17
18 // Компоненти можуть розширювати інші компоненти.
19 class Circle extends Dot is
20     field radius
21
22 constructor Circle(x, y, radius) { ... }
23
24 method draw() is
25     // Намалювати коло в координатах X, Y з радіусом R.
26
27

```

```
28 // Контейнер містить операції додавання/видалення дочірніх
29 // компонентів. Усі стандартні операції інтерфейсу компонентів
30 // він делегує кожному з дочірніх компонентів.
31 class CompoundGraphic implements Graphic is
32     field children: array of Graphic
33
34     method add(child: Graphic) is
35         // Додати компонент до списку дочірніх.
36
37     method remove(child: Graphic) is
38         // Прибрати компонент зі списку дочірніх.
39
40     method move(x, y) is
41         foreach (child in children) do
42             child.move(x, y)
43
44     method draw() is
45         // 1. Для кожного дочірнього компонента:
46         //      – Відобразити компонент.
47         //      – Визначити координати максимальної межі.
48         // 2. Намалювати пунктирну межу навколо всієї області.
49
50     // Програма працює одноманітно, як з одиничними компонентами,
51     // так і з цілими групами компонентів.
52 class ImageEditor is
53     field all: CompoundGraphic
54
55     method load() is
56         all = new CompoundGraphic()
57         all.add(new Dot(1, 2))
58         all.add(new Circle(5, 3, 10))
59         // ...
```

```

60 // Групування обраних компонентів в один складний компонент.
61 method groupSelected(components: array of Graphic) is
62     group = new CompoundGraphic()
63     foreach (component in components) do
64         group.add(component)
65         all.remove(component)
66     all.add(group)
67     // Усі компоненти будуть промальованими.
68     all.draw()

```

## Застосування

 Якщо вам потрібно представити деревоподібну структуру об'єктів.

 Патерн Компонувальник пропонує зберігати в складових об'єктах посилання на інші прості або складові об'єкти. Вони, у свою чергу, теж можуть зберігати свої вкладені об'єкти і так далі. У підсумку, ви можете будувати складну деревоподібну структуру даних, використовуючи всього два основних різновиди об'єктів.

 Якщо клієнти повинні однаково трактувати прості та складові об'єкти.

 Завдяки тому, що прості та складові об'єкти реалізують спільний інтерфейс, клієнту байдуже, з яким саме об'єктом він працюватиме.

## Кроки реалізації

1. Переконайтесь, що вашу бізнес-логіку можна представити як деревоподібну структуру. Спробуйте розбити її на прості компоненти й контейнери. Пам'ятайте, що контейнери можуть містити як прості компоненти, так і інші вкладені контейнери.
2. Створіть загальний інтерфейс компонентів, який об'єднає операції контейнерів та простих компонентів дерева. Інтерфейс буде вдалим, якщо ви зможете використовувати його, щоб взаємозамінити прості й складові компоненти без втрати сенсу.
3. Створіть клас компонентів-листя, які не мають подальших відгалужень. Майте на увазі, що програма може містити декілька таких класів.
4. Створіть клас компонентів-контейнерів і додайте до нього масив для зберігання посилань на вкладені компоненти. Цей масив повинен бути здатен містити як прості, так і складові компоненти, тому переконайтесь, що його оголошено з типом інтерфейсу компонентів.

Реалізуйте в контейнері методи інтерфейсу компонентів, пам'ятаючи про те, що контейнери повинні делегувати основну роботу своїм дочірнім компонентам.

5. Додайте операції додавання й видалення дочірніх компонентів до класу контейнерів.

Майте на увазі, що методи додавання/видалення дочірніх компонентів можна оголосити також і в інтерфейсі компонентів. Так, це порушить *принцип розділення інтерфейсу*, тому що реалізації методів будуть порожніми в компонентах-листях. Проте усі компоненти дерева стануть дійсно однаковими для клієнта.

## **Переваги та недоліки**

- ✓ Спрошує архітектуру клієнта при роботі зі складним деревом компонентів.
- ✓ Полегшує додавання нових видів компонентів.
- ✗ Створює занадто загальний дизайн класів.

## **Відносини з іншими патернами**

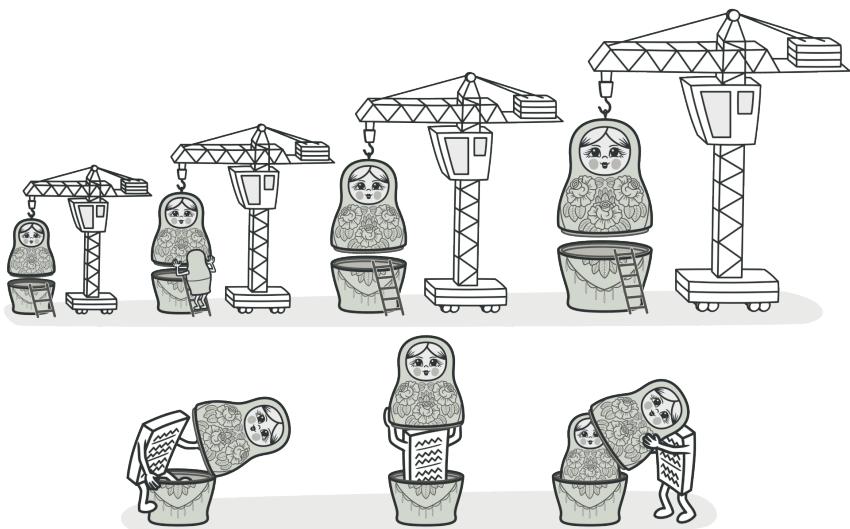
- **Будівельник** дозволяє покроково конструювати дерево **Компонувальника**.
- **Ланцюжок обов'язків** часто використовують разом з **Компонувальником**. У цьому випадку запит передається від дочірніх компонентів до їхніх батьків.
- Ви можете обходити дерево **Компонувальника**, використовуючи **Ітератор**.

- Ви можете виконати якусь дію над усім деревом **Компонувальника** за допомогою **Відвідувача**.
- **Компонувальник** часто поєднують з **Легковаговиком**, щоб реалізувати спільні гілки дерева та заощадити при цьому пам'ять.
- **Компонувальник** та **Декоратор** мають схожі структури класів, бо обидва побудовані на рекурсивній вкладеності. Вона дозволяє зв'язати в одну структуру нескінченну кількість об'єктів.

*Декоратор* обгортає тільки один об'єкт, а вузол *Компонувальника* може мати багато дітей. *Декоратор* додає вкладеному об'єкту нової функціональності, а *Компонувальник* не додає нічого нового, але «підсумовує» результати всіх своїх дітей.

Але вони можуть і співпрацювати: *Компонувальник* може використовувати *Декоратор*, щоб перевизначити функції окремих частин дерева компонентів.

- Архітектура, побудована на **Компонувальниках** та **Декораторах**, часто може поліпшуватися за рахунок впровадження **Прототипу**. Він дозволяє клонувати складні структури об'єктів, а не збирати їх заново.



# ДЕКОРАТОР

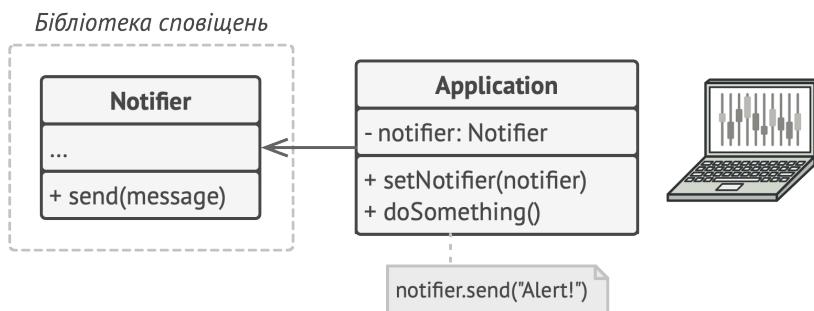
Також відомий як: *Wrapper, Обгортка, Decorator*

**Декоратор** – це структурний патерн проектування, що дає змогу динамічно додавати об'єктам нову функціональність, загортуючи їх у корисні «обгортки».

## (:() Проблема

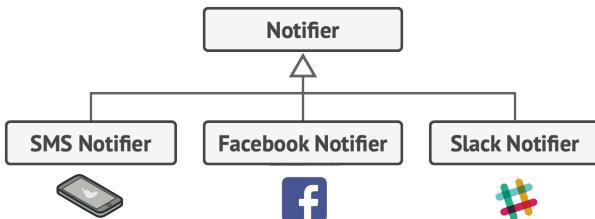
Ви працюєте над бібліотекою сповіщень, яку можна підключати до різноманітних програм, щоб отримувати сповіщення про важливі події.

Основою бібліотеки є клас `Notifier` з методом `send`, який приймає на вхід рядок-повідомлення і надсилає його всім адміністраторам електронною поштою. Стороння програма повинна створити й налаштувати цей об'єкт, вказавши, кому надсилати сповіщення, та використовувати його щоразу, коли щось відбувається.



*Сторонні програми використовують головний клас сповіщень.*

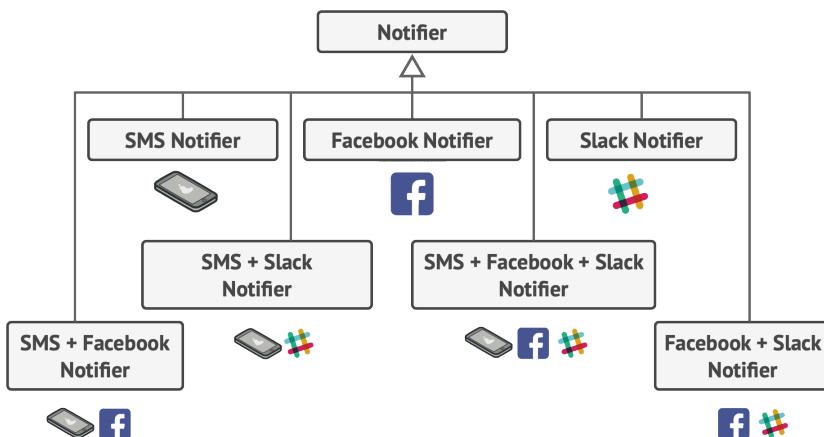
В якийсь момент стало зрозуміло, що користувачам не вистачає одних тільки email-сповіщень. Деякі з них хотіли б отримувати сповіщення про критичні проблеми через SMS. Інші хотіли б отримувати їх у вигляді Facebook-повідомлень. Корпоративні користувачі хотіли би бачити повідомлення у Slack.



*Кожен тип сповіщення живе у власному підкласі.*

Спершу ви додали кожен з типів сповіщень до програми, успадкувавши їх від базового класу `Notifier`. Тепер користувачі могли вибрати один з типів сповіщень, який і використовувався надалі.

Але потім хтось резонно запитав, чому не можна увімкнути кілька типів сповіщень одночасно? Адже, якщо у вашому будинку раптом почалася пожежа, ви б хотіли отримати сповіщення по всіх каналах, чи не так?



*Комбінаторний вибух підкласів при поєднанні типів сповіщень.*

Ви зробили спробу реалізувати всі можливі комбінації підкласів сповіщень, але після того, як додали перший десяток класів, стало зрозуміло, що такий підхід неймовірно роздуває код програми.

Отже, потрібен інший спосіб комбінування поведінки об'єктів, який не призводить до збільшення кількості підкласів.

## Рішення

Спадкування – це перше, що приходить в голову багатьом програмістам, коли потрібно розширити яку-небудь чинну поведінку. Проте механізм спадкування має кілька прикрих проблем.

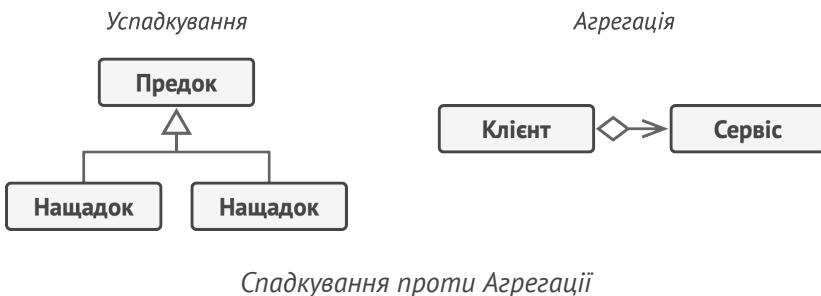
- Він **статичний**. Ви не можете змінити поведінку об'єкта, який вже існує. Для цього необхідно створити новий об'єкт, вибравши інший підклас.
- Він **не дозволяє наслідувати поведінку декількох класів одночасно**. Тому доведеться створювати безліч підкласів-комбінацій, щоб досягти поєднання поведінки.

Одним зі способів, що дозволяє обійти ці проблеми, є заміна спадкування *агрегацією* або *композицією*<sup>1</sup>. Це той випадок, коли один об'єкт утримує інший і делегує йому роботу,

---

1. Композиція – це більш суворий варіант агрегації, при якому компоненти не можуть існувати без контейнера.

замість того, щоб самому успадковувати його поведінку. Саме на цьому принципі побудовано патерн Декоратор.

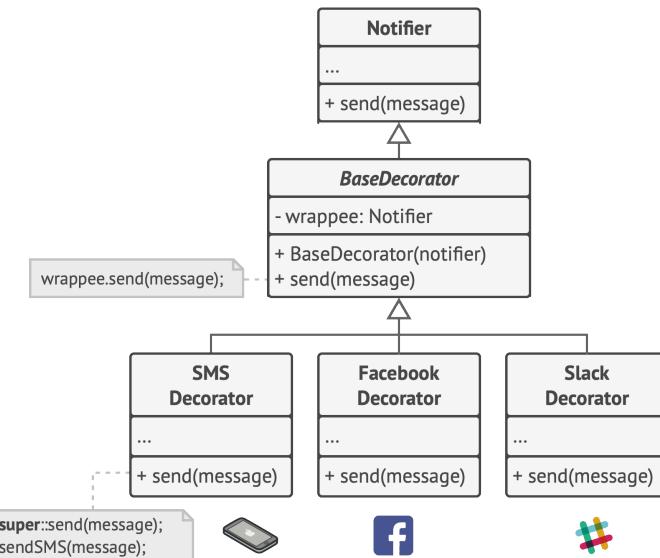


Декоратор має альтернативну назву – *обгортка*. Вона більш вдало описує суть патерна: ви розміщуєте цільовий об'єкт у іншому об'єкті-обгортці, який запускає базову поведінку об'єкта, а потім додає до результату щось своє.

Обидва об'єкти мають загальний інтерфейс, тому для користувача немає жодної різниці, з чим працювати – з чистим чи загорнутим об'єктом. Ви можете використовувати кілька різних обгорток одночасно – результат буде мати об'єднану поведінку всіх обгорток.

В нашому прикладі зі сповіщеннями залишимо в базовому класі просте надсилання сповіщень електронною поштою, а розширені способи зробимо декораторами.

Стороння програма, яка виступає клієнтом, під час початкового налаштовування буде загортати об'єкт сповіщення в ті обгортки, які відповідають бажаному способу сповіщення.



Розширені способи надсилання сповіщень стають декораторами.

```

stack = new Notifier()
if (facebookEnabled)
    stack = new FacebookDecorator(stack)
if (slackEnabled)
    stack = new SlackDecorator(stack)

app.setNotifier(stack)
  
```



```

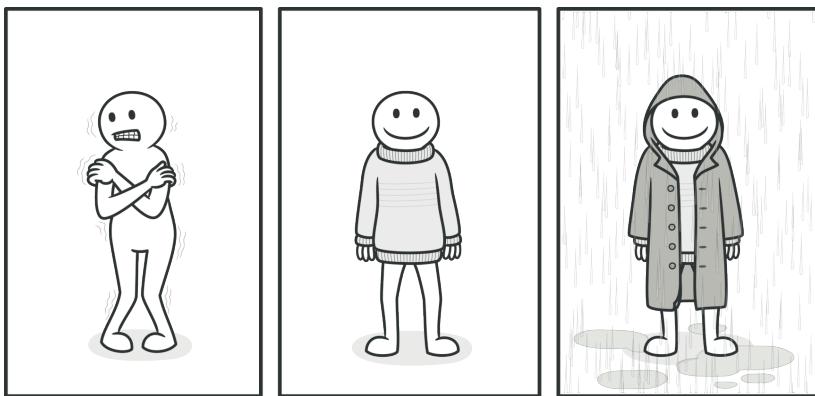
notifier.send("Alert!")
// Email → Facebook → Slack
  
```

Програма може збирати складові об'єкти з декораторів.

Остання обгортка у списку буде саме тим об'єктом, з яким клієнт працюватиме увесь інший час. Для решти клієнтського коду нічого не зміниться, адже всі обгортки мають такий самий інтерфейс, що і базовий клас сповіщень.

Так само можна змінювати не тільки спосіб доставки сповіщень, але й форматування, список адресатів і так далі. До того ж клієнт зможе «дозагорнути» об'єкт у будь-які інші обгортки, якщо йому цього захочеться.

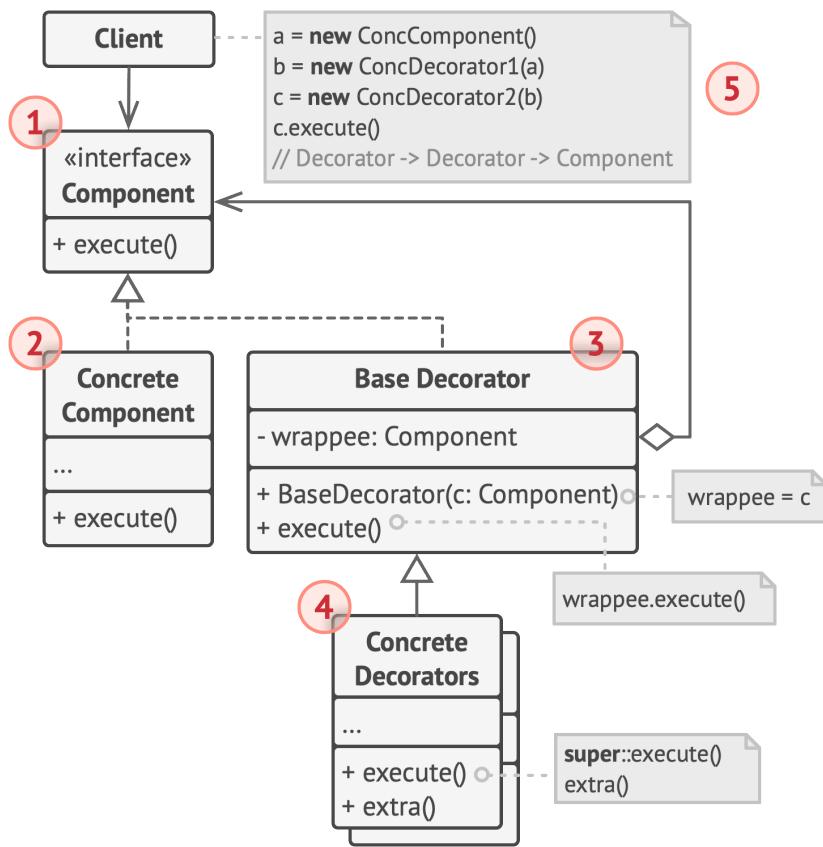
## Аналогія з життя



*Одяг можна одягати кількома шарами, отримуючи комбінований ефект.*

Будь-який одяг – це аналог Декоратора. Застосовуючи Декоратор, ви не змінюєте початковий клас і не створюєте дочірніх класів. Так само з одягом: вдягаючи светра, ви не перестаєте бути собою, але отримуєте нову властивість – захист від холоду. Ви можете піти далі й одягти зверху ще один декоратор – плащ, щоб захиститися від дощу.

## Структура



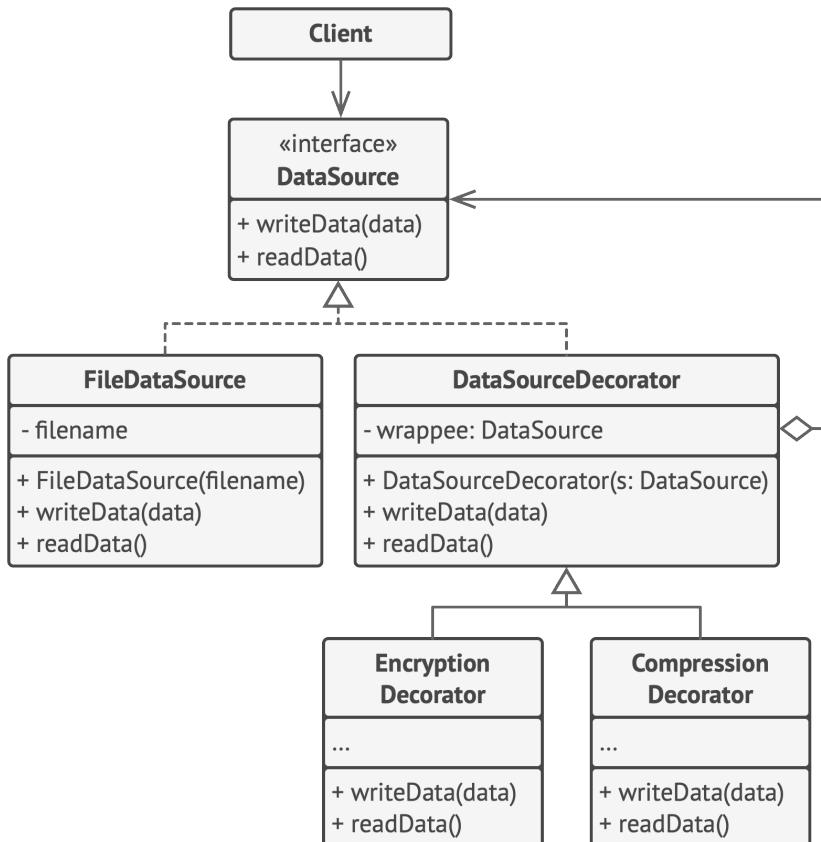
1. **Компонент** задає загальний інтерфейс обгорток та об'єктів, що загортуються.
2. **Конкретний компонент** визначає клас об'єктів, що загортуються. Він містить якусь базову поведінку, яку потім змінюють декоратори.

3. **Базовий декоратор** зберігає посилання на вкладений об'єкт-компонент. Це може бути як конкретний компонент, так і один з конкретних декораторів. Базовий декоратор делегує всі свої операції вкладеному об'єкту. Додаткова поведінка житиме в конкретних декораторах.
4. **Конкретні декоратори** – це різні варіації декораторів, що містять додаткову поведінку. Вона виконується до або після виклику аналогічної поведінки загорнутого об'єкта.
5. **Клієнт** може обертати прості компоненти й декоратори в інші декоратори, працюючи з усіма об'єктами через загальний інтерфейс компонентів.

## # Псевдокод

У цьому прикладі **Декоратор** захищає фінансові дані додатковими рівнями безпеки прозоро для коду, який їх використовує.

Програма обгортав клас даних у шифруючу та стискаючу обгортку, які при читанні видають оригінальні дані, а при записі – зашифровані та стислі.



*Приклад шифрування й компресії даних за допомогою обгорток.*

Декоратори, як і сам клас даних, мають спільний інтерфейс. Тому клієнтському коду не важливо, з чим працювати – зі звичайним об'єктом даних чи з загорнутим.

```

1 // Загальний інтерфейс компонентів.
2 interface DataSource is
3     method writeData(data)
4     method readData():data
  
```

```
5 // Один з конкретних компонентів реалізує базову
6 // функціональність.
7 class FileDataSource implements DataSource is
8     constructor FileDataSource(filename) { ... }
9
10    method writeData(data) is
11        // Записати дані до файлу.
12
13    method readData():data is
14        // Прочитати дані з файлу.
15
16 // Базовий клас усіх декораторів містить код обгортування.
17 class DataSourceDecorator implements DataSource is
18     protected field wrappee: DataSource
19
20     constructor DataSourceDecorator(source: DataSource) is
21         wrappee = source
22
23     method writeData(data) is
24         wrappee.writeData(data)
25
26     method readData():data is
27         return wrappee.readData()
28
29 // Конкретні декоратори додають щось своє до базової поведінки
30 // обгорнутого компонента.
31 class EncryptionDecorator extends DataSourceDecorator is
32     method writeData(data) is
33         // 1. Зашифрувати подані дані.
34         // 2. Передати зашифровані дані до методу writeData
35         // обгорнутого об'єкта (wrappee).
36
```

```

37 method readData():data is
38     // 1. Отримати дані з методу readData обгорнутого
39     // об'єкта (wrappee).
40     // 2. Розшифрувати їх, якщо вони зашифровані.
41     // 3. Повернути результат.
42
43 // Декорувати можна не тільки базові компоненти, але й вже
44 // обгорнуті об'єкти.
45 class CompressionDecorator extends DataSourceDecorator is
46     method writeData(data) is
47         // 1. Запакувати подані дані.
48         // 2. Передати запаковані дані до методу writeData
49         // обгорнутого об'єкта (wrappee).
50
51     method readData():data is
52         // 1. Отримати дані з методу readData обгорнутого
53         // об'єкта (wrappee).
54         // 2. Розпакувати їх, якщо вони запаковані.
55         // 3. Повернути результат.
56
57
58 // Варіант 1. Простий приклад збирання та використання
59 // декораторів.
60 class Application is
61     method dumbUsageExample() is
62         source = new FileDataSource("somefile.dat")
63         source.writeData(salaryRecords)
64         // До файлу було занесено чисті дані.
65
66         source = new CompressionDecorator(source)
67         source.writeData(salaryRecords)
68         // До файлу було занесено стислі дані.

```

```

69     source = new EncryptionDecorator(source)
70     // Зараз у source знаходиться зв'язка з трьох об'єктів:
71     // Encryption > Compression > FileDataSource
72
73     source.writeData(salaryRecords)
74     // До файлу було занесено стислі та зашифровані дані.
75
76
77 // Варіант 2. Клієнтський код, який використовує зовнішнє
78 // джерело даних. Клас SalaryManager нічого не знає про те, як
79 // саме буде зчитано та записано дані. Він отримує вже готове
80 // джерело даних.
81 class SalaryManager is
82     field source: DataSource
83
84     constructor SalaryManager(source: DataSource) { ... }
85
86     method load() is
87         return source.readData()
88
89     method save() is
90         source.writeData(salaryRecords)
91         // ...Інші корисні методи...
92
93
94 // Програма може різним шляхом збирати об'єкти, які декоруються
95 // залежно від умов використання.
96 class ApplicationConfigurator is
97     method configurationExample() is
98         source = new FileDataSource("salary.dat")
99         if (enabledEncryption)
100             source = new EncryptionDecorator(source)

```

```

101     if (enabledCompression)
102         source = new CompressionDecorator(source)
103
104     logger = new SalaryManager(source)
105     salary = logger.load()
106     // ...

```

## Застосування

-  Якщо вам потрібно додавати об'єктам нові обов'язки «на льоту», непомітно для коду, який їх використовує.
-  Об'єкти вкладаються в обгортки, які мають додаткові поведінки. Обгортки і самі об'єкти мають одинаковий інтерфейс, тому клієнтам не важливо, з чим працювати – зі звичайним об'єктом чи з загорнутим.
-  Якщо не можна розширити обов'язки об'єкта за допомогою спадкування.
-  У багатьох мовах програмування є ключове слово `final`, яке може заблокувати спадкування класу. Розширити такі класи можна тільки за допомогою Декоратора.

## Кроки реалізації

1. Переконайтесь, що у вашому завданні присутні основний компонент і декілька опціональних доповнень-надбудов над ним.
2. Створіть інтерфейс компонента, який описував би загальні методи як для основного компонента, так і для його доповнень.
3. Створіть клас конкретного компонента й помістіть в нього основну бізнес-логіку.
4. Створіть базовий клас декораторів. Він повинен мати поле для зберігання посилань на вкладений об'єкт-компонент. Усі методи базового декоратора повинні делегувати роботу вкладеному об'єкту.
5. Конкретний компонент, як і базовий декоратор, повинні дотримуватися одного і того самого інтерфейсу компонента.
6. Створіть класи конкретних декораторів, успадковуючи їх від базового декоратора. Конкретний декоратор повинен виконувати свою додаткову функціональність, а потім (або перед цим) викликати цю ж операцію загортаного об'єкта.
7. Клієнт бере на себе відповідальність за конфігурацію і порядок загортання об'єктів.

## ΔΔ Переваги та недоліки

- ✓ Більша гнучкість, ніж у спадкування.
- ✓ Дозволяє додавати обов'язки «на льоту».
- ✓ Можна додавати кілька нових обов'язків одразу.
- ✓ Дозволяє мати кілька дрібних об'єктів, замість одного об'єкта «на всі випадки життя».
- ✗ Важко конфігурувати об'єкти, які загорнуто в декілька обгорток одночасно.
- ✗ Велика кількість крихітних класів.

## ↔ Відносини з іншими патернами

- **Адаптер** змінює інтерфейс існуючого об'єкта. **Декоратор** покращує інший об'єкт без зміни його інтерфейсу. Причому **Декоратор** підтримує рекурсивну вкладуваність, на відміну від **Адаптеру**.
- **Адаптер** надає класу альтернативний інтерфейс. **Декоратор** надає розширений інтерфейс. **Замісник** надає той самий інтерфейс.
- **Ланцюжок обов'язків** та **Декоратор** мають дуже схожі структури. Обидва патерни базуються на принципі рекурсивного виконання операції через серію пов'язаних об'єктів. Але є декілька важливих відмінностей.

Обробники в *Ланцюжку обов'язків* можуть виконувати довільні дії, незалежні одна від одної, а також у будь-який момент переривати подальшу передачу ланцюжком. З іншого боку, *Декоратори* розширяють певну дію, не ламаючи інтерфейс базової операції і не перериваючи виконання інших декораторів.

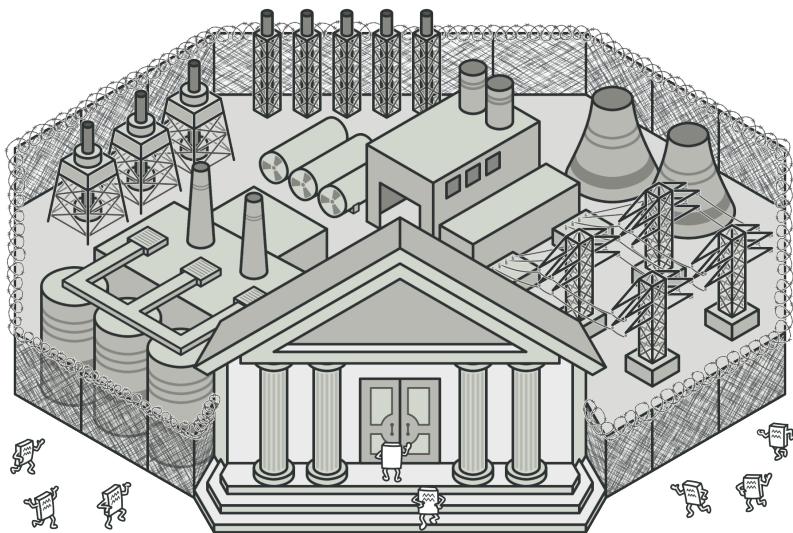
- **Компонувальник** та **Декоратор** мають схожі структури класів, бо обидва побудовані на рекурсивній вкладеності. Вона дозволяє зв'язати в одну структуру нескінченну кількість об'єктів.

*Декоратор* обгортає тільки один об'єкт, а вузол *Компонувальника* може мати багато дітей. *Декоратор* додає вкладеному об'єкту нової функціональності, а *Компонувальник* не додає нічого нового, але «підсумовує» результати всіх своїх дітей.

Але вони можуть і співпрацювати: *Компонувальник* може використовувати *Декоратор*, щоб перевизначити функції окремих частин дерева компонентів.

- Архітектура, побудована на **Компонувальниках** та **Декораторах**, часто може поліпшуватися за рахунок впровадження **Прототипу**. Він дозволяє клонувати складні структури об'єктів, а не збирати їх заново.
- **Стратегія** змінює поведінку об'єкта «зсередини», а **Декоратор** змінює його «ззовні».

- **Декоратор** та **Замісник** мають схожі структури, але різні призначення. Вони схожі тим, що обидва побудовані на композиції та делегуванні роботи іншому об'єкту. Патерни відрізняються тим, що **Замісник** сам керує життям сервісного об'єкта, а обортання **Декораторів** контролюється клієнтом.



# ФАСАД

Також відомий як: *Facade*

**Фасад** – це структурний патерн проектування, який надає простий інтерфейс до складної системи класів, бібліотеки або фреймворку.

## **Проблема**

Вашому коду доводиться працювати з великою кількістю об'єктів певної складної бібліотеки чи фреймворка. Ви повинні самостійно ініціалізувати ці об'єкти, стежити за правильним порядком залежностей тощо.

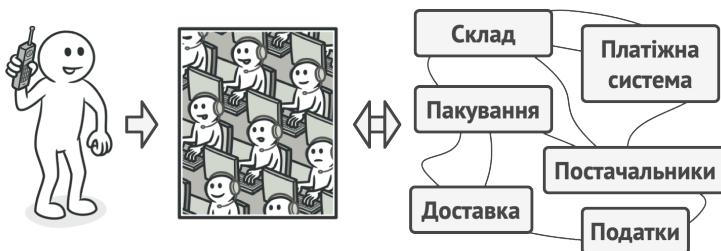
В результаті бізнес-логіка ваших класів тісно переплітається з деталями реалізації сторонніх класів. Такий код досить складно розуміти та підтримувати.

## **Рішення**

Фасад – це простий інтерфейс для роботи зі складною підсистемою, яка містить безліч класів. Фасад може бути спрощеним відображенням системи, що не має 100% тієї функціональності, якої можна було б досягти, використовуючи складну підсистему безпосередньо. Разом з тим, він надає саме ті «фічі», які потрібні клієнтові, і приховує все інше.

Фасад корисний у тому випадку, якщо ви використовуєте якусь складну бібліотеку з безліччю рухомих частин, з яких вам потрібна тільки частина. Наприклад, програма, що заливає в соціальні мережі відео з кошенятками, може використовувати професійну бібліотеку для стискання відео, але все, що потрібно клієнтському коду цієї програми, – це простий метод `encode(filename, format)`. Створивши клас з таким методом, ви реалізуєте свій перший фасад.

## 🚗 Аналогія з життя

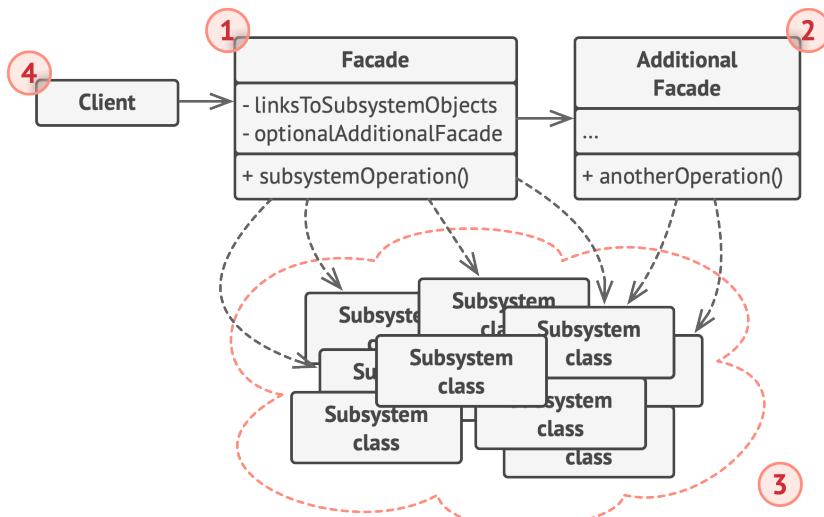


Приклад замовлення через телефон.

Коли ви телефонуєте до магазину і робите замовлення, співробітник служби підтримки є вашим фасадом до всіх служб і відділів магазину. Він надає вам спрощений інтерфейс до системи створення замовлення, платіжної системи та відділу доставки.

## ---

## Структура



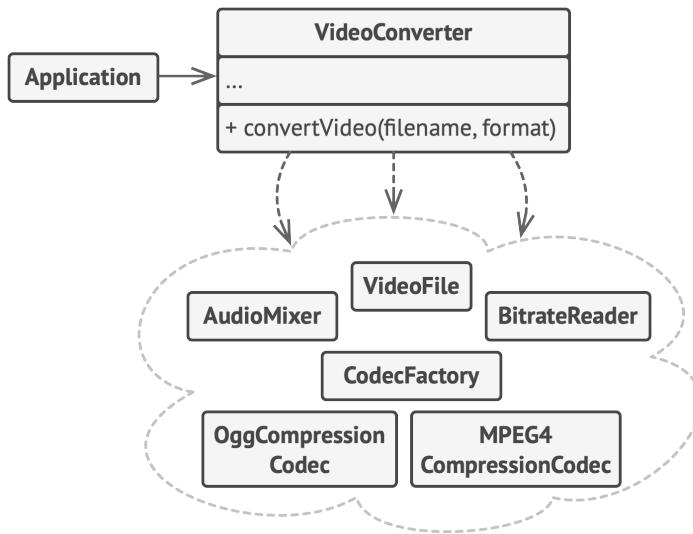
1. **Фасад** надає швидкий доступ до певної функціональності підсистеми. Він «знає», яким класам потрібно переадресувати запит, і які дані для цього потрібні.
2. **Додатковий фасад** можна ввести, щоб не захаращувати єдиний фасад різнопідсистемною функціональністю. Він може використовуватися як клієнтом, так й іншими фасадами.
3. **Складна підсистема** має безліч різноманітних класів. Для того, щоб примусити усіх їх щось робити, потрібно знати подробиці влаштування підсистеми, порядок ініціалізації об'єктів та інші деталі.

Класи підсистеми не знають про існування фасаду і працюють один з одним безпосередньо.

4. **Клієнт** використовує фасад замість безпосередньої роботи з об'єктами складної підсистеми.

## # Псевдокод

У цьому прикладі **Фасад** спрощує роботу зі складним фреймворком конвертації відео.



Приклад ізоляції множини залежностей в одному фасаді.

Замість безпосередньої роботи з дюжиною класів, фасад надає коду програми єдиний метод для конвертації відео, який сам подбає про те, щоб правильно налаштувати потрібні об'єкти фреймворку і отримати необхідний результат.

```

1 // Класи складного стороннього фреймворку конвертації відео. Ми
2 // не контролюємо цей код, тому не можемо його спростити.
3 class VideoFile
4 // ...
5
6 class OggCompressionCodec
7 // ...
8
9 class MPEG4CompressionCodec
10 // ...

```

```
11 class CodecFactory
12 // ...
13
14 class BitrateReader
15 // ...
16
17 class AudioMixer
18 // ...
19
20
21 // Замість цього, ми створюємо Фасад – простий інтерфейс для
22 // роботи зі складним фреймворком. Фасад не має всієї
23 // функціональності фреймворку, але приховує його складність від
24 // клієнтів.
25 class VideoConverter is
26     method convert(filename, format):File is
27         file = new VideoFile(filename)
28         sourceCodec = new CodecFactory.extract(file)
29         if (format == "mp4")
30             destinationCodec = new MPEG4CompressionCodec()
31         else
32             destinationCodec = new OggCompressionCodec()
33         buffer = BitrateReader.read(filename, sourceCodec)
34         result = BitrateReader.convert(buffer, destinationCodec)
35         result = (new AudioMixer()).fix(result)
36         return new File(result)
37
38 // Програма не залежить від складного фреймворку конвертації
39 // відео. До речі, якщо ви раптом вирішите змінити фреймворк,
40 // вам потрібно буде переписати тільки клас фасаду.
41 class Application is
42     method main() is
```

```

43     convertor = new VideoConverter()
44     mp4 = convertor.convert("funny-cats-video.ogg", "mp4")
45     mp4.save()

```

## Застосування

 Якщо вам потрібно надати простий або урізаний інтерфейс до складної підсистеми.

 Часто підсистеми ускладнюються в міру розвитку програми. Застосування більшості патернів призводить до появи менших класів, але у великій кількості. Таку підсистему простіше використовувати повторно, налаштовуючи її кожен раз під конкретні потреби, але, разом з тим, використовувати таку підсистему без налаштування важче. Фасад пропонує певний вид системи за замовчuvанням, який влаштовує більшість клієнтів.

 Якщо ви хочете розкласти підсистему на окремі рівні.

 Використовуйте фасади для визначення точок входу на кожен рівень підсистеми. Якщо підсистеми залежать одна від одної, тоді залежність можна спростити, дозволивши підсистемам обмінюватися інформацією тільки через фасади.

Наприклад, візьмемо ту ж саму складну систему конвертації відео. Ви хочете розбити її на окремі шари для роботи з аудіо й відео. Можна спробувати створити фасад для кожної

з цих частин і примусити класи аудіо та відео обробки спілкуватися один з одним через ці фасади, а не безпосередньо.

## Кроки реалізації

1. Визначте, чи можна створити більш простий інтерфейс, ніж той, який надає складна підсистема. Ви на правильному шляху, якщо цей інтерфейс позбавить клієнта від необхідності знати подrobiці підсистеми.
2. Створіть клас фасаду, що реалізує цей інтерфейс. Він повинен переадресовувати виклики клієнта потрібним об'єктам підсистеми. Фасад повинен буде подбати про те, щоб правильно ініціалізувати об'єкти підсистеми.
3. Ви отримаєте максимум користі, якщо клієнт працюватиме тільки з фасадом. В такому випадку зміни в підсистемі стосуватимуться тільки коду фасаду, а клієнтський код залишиться робочим.
4. Якщо відповідальність фасаду стає розмитою, подумайте про введення додаткових фасадів.

## Переваги та недоліки

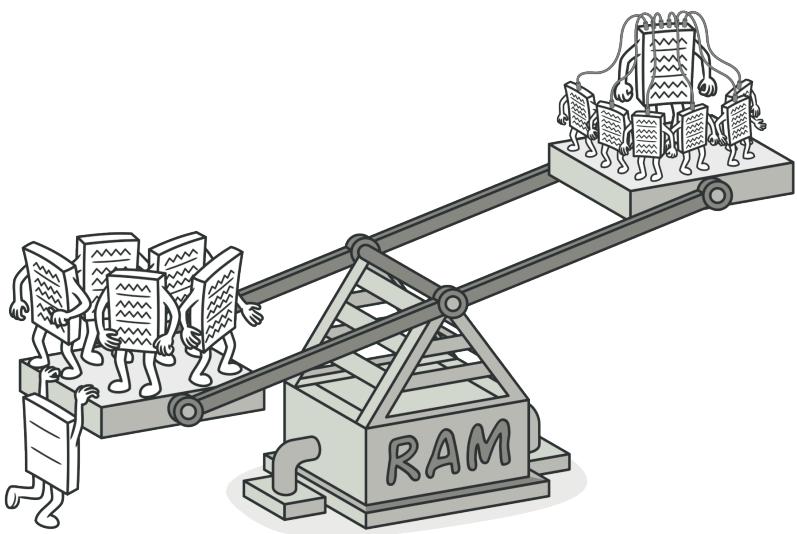
- ✓ Ізоляє клієнтів від компонентів складної підсистеми.

- ✗ Фасад ризикує стати **божественим об'єктом**, прив'язаним до всіх класів програми.

## ↔ Відносини з іншими патернами

- **Фасад** задає новий інтерфейс, тоді як **Адаптер** повторно використовує старий. *Адаптер* обгортає тільки один клас, а *Фасад* обгортає цілу підсистему. Крім того, *Адаптер* дозволяє двом існуючим інтерфейсам працювати спільно, замість того, щоб визначити повністю новий.
- **Абстрактна фабрика** може бути використана замість **Фасаду** для того, щоб приховати платформо-залежні класи.
- **Легковаговик** показує, як створювати багато дрібних об'єктів, а **Фасад** показує, як створити один об'єкт, який відображає цілу підсистему.
- **Посередник** та **Фасад** схожі тим, що намагаються організувати роботу багатьох існуючих класів.
  - *Фасад* створює спрощений інтерфейс підсистеми, не вносячи в неї жодної додаткової функціональності. Сама підсистема не знає про існування *Фасаду*. Класи підсистеми спілкуються один з одним безпосередньо.
  - *Посередник* централізує спілкування між компонентами системи. Компоненти системи знають тільки про існування *Посередника*, у них немає прямого доступу до інших компонентів.

- **Фасад** можна зробити **Одннаком**, оскільки зазвичай потрібен тільки один об'єкт-фасад.
- **Фасад** схожий на **Замісник** тим, що замінює складну підсистему та може сам її ініціалізувати. Але, на відміну від **Фасаду**, **Замісник** має такий самий інтерфейс, що і його службовий об'єкт, завдяки чому їх можна взаємозамінити.



# ЛЕГКОВАГОВИК

Також відомий як: *Пристосуванець, Kew, Flyweight*

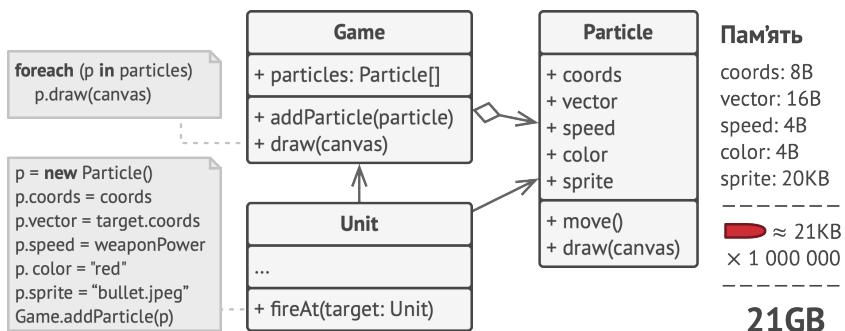
**Легковаговик** – це структурний патерн проектування, що дає змогу вмістити більшу кількість об'єктів у відведеній оперативній пам'яті. Легковаговик заощаджує пам'ять, розподіляючи спільний стан об'єктів між собою, замість зберігання одинакових даних у кожному об'єкті.

## (:() Проблема

На дозвіллі ви вирішили написати невелику гру, в якій гравці переміщаються по карті та стріляють один в одного. Фішкою гри повинна була стати реалістична система частинок. Кулі, снаряди, уламки від вибухів – все це повинно реалістично літати та гарно виглядати.

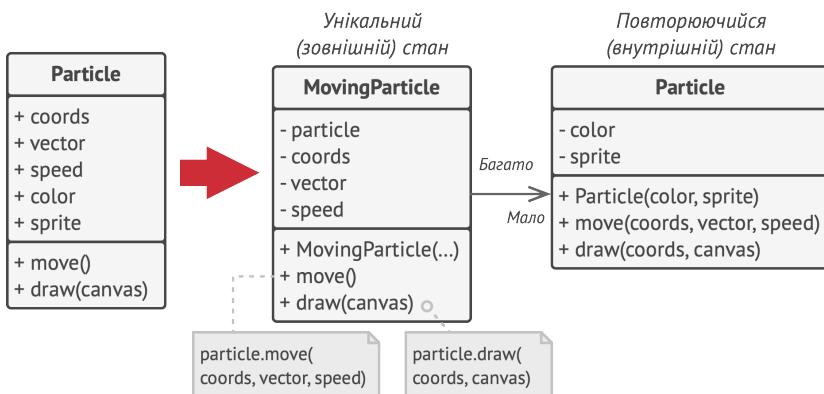
Гра відмінно працювала на вашому потужному комп’ютері, проте ваш друг повідомив, що гра починає гальмувати й вилітає через кілька хвилин після запуску. Передивившись логи, ви виявили, що гра вилітає через нестачу оперативної пам’яті. У вашого друга комп’ютер значно менше «прокачаний», тому проблема в нього й проявляється так швидко.

Дійсно, кожна частинка у грі представлена власним об’єктом, що має безліч даних. У певний момент, коли побоїще на екрані досягає кульмінації, оперативна пам’ять комп’ютера вже не може вмістити нові об’єкти частинок, і програма «вилітає».



## 😊 Рішення

Якщо уважно подивитися на клас частинок, то можна помітити, що колір і спрайт займають найбільше пам'яті. Більше того, ці поля зберігаються в кожному об'єкті, хоча фактично їхні значення є однаковими для більшості частинок.

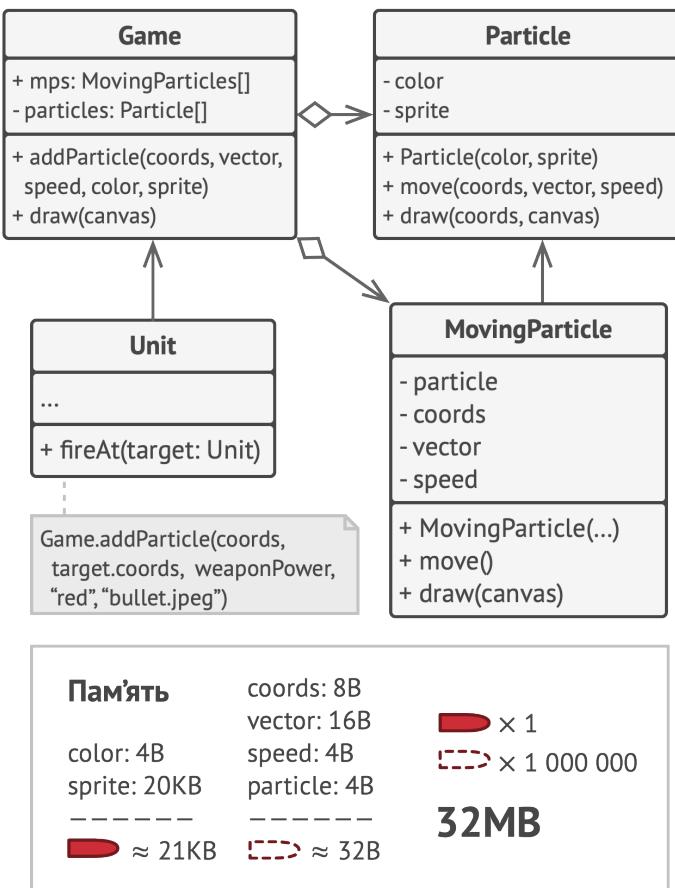


Інший стан об'єктів – координати, вектор руху й швидкість відрізняються для всіх частинок. Таким чином, ці поля можна розглядати як контекст, у якому використовується частинка, а колір і спрайт – це дані, що не змінюються в часі.

Незмінні дані об'єкта прийнято називати «внутрішнім станом». Всі інші дані – це «зовнішній стан».

Патерн Легковаговик пропонує не зберігати зовнішній стан у класі, а передавати його до тих чи інших методів через параметри. Таким чином, одні і ті самі об'єкти можна буде повторно використовувати в різних контекстах. Головна ж

перевага в тому, що тепер знадобиться набагато менше об'єктів, адже вони тепер відрізняються тільки внутрішнім станом, а він не має так багато варіацій.



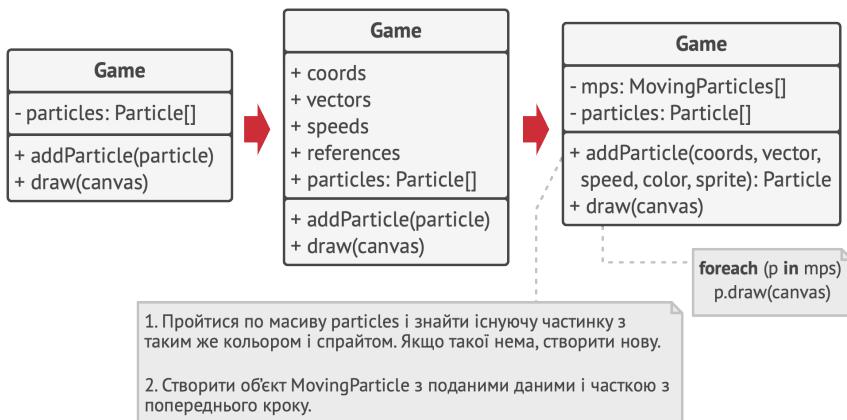
У нашому прикладі з частинками достатньо буде залишити лише три об'єкти, що відрізняються спрайтами і кольором – для куль, снарядів та уламків. Нескладно здогадатися, що такі полегшені об'єкти називають *легковаговиками*<sup>1</sup>.

1. Назва прийшла з боксу і означає вагову категорію до 50 кг.

## Сховище зовнішнього стану

Але куди переїде зовнішній стан? Адже хтось повинен його зберігати. Найчастіше його переміщують до контейнера, який керував об'єктами до застосування патерна.

В нашому випадку це був головний клас гри. Ви могли б додати до його класу поля-масиви для зберігання координат, векторів і швидкостей частинок. Крім цього, потрібен буде ще один масив для зберігання посилань на об'єкти-легковаговики, що відповідають тій чи іншій частинці.



Більш елегантним рішенням було б створити додатковий клас-контекст, який пов'язував би зовнішній стан з тим чи іншим легковаговиком. Це дозволить обійтися тільки одним полем-масивом у класі контейнера.

«Але стривайте, нам буде потрібно стільки ж цих об'єктів, скільки було на самому початку!» — скажете ви і будете

праві! Але річ у тім, що об'єкти-контексти займають набагато менше місця, ніж початкові. Адже найважчі поля залишилися всередині легковаговика (вибачте за каламбур), і зараз ми будемо посилатися на ці об'єкти з контекстів, замість того, щоб повторно зберігати стан, що дублюється.

## **Незмінність Легковаговиків**

Оскільки об'єкти легковаговиків будуть використані в різних контекстах, ви повинні бути впевненими в тому, що їхній стан неможливо змінити після створення. Весь внутрішній стан легковаговик повинен отримувати через параметри конструктора. Він не повинен мати сеттерів і публічних полів.

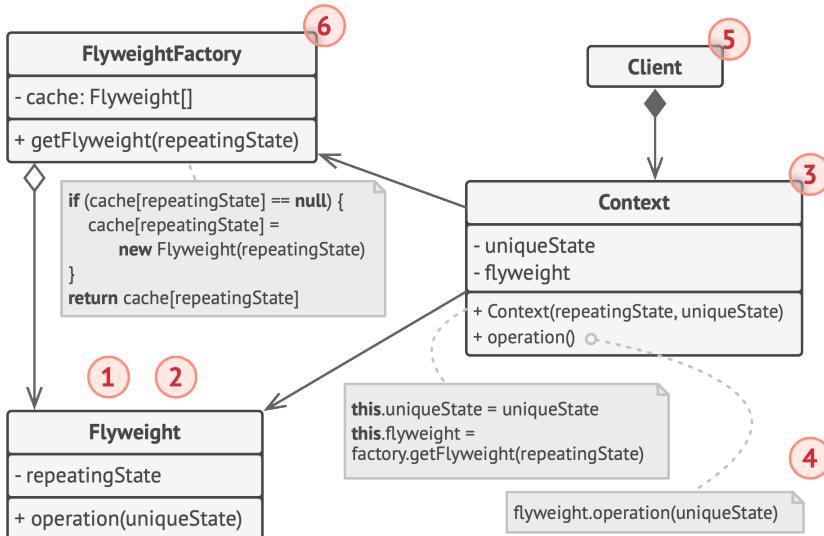
## **Фабрика Легковаговиків**

Для зручності роботи з легковаговиками і контекстами можна створити фабричний метод, що приймає в параметрах увесь внутрішній (іноді й зовнішній) стан бажаного об'єкта.

Найбільша користь цього методу в тому, щоб знаходити вже створених легковаговиків з таким самим внутрішнім станом, як потрібно. Якщо легковаговик знаходиться, його можна повторно використовувати. Якщо немає — просто створюємо новий.

Зазвичай цей метод додають до контейнера легковаговиків або створюють окремий клас-фабрику. Його навіть можна зробити статичним і розмістити в класі легковаговиків.

## STRUCTURE



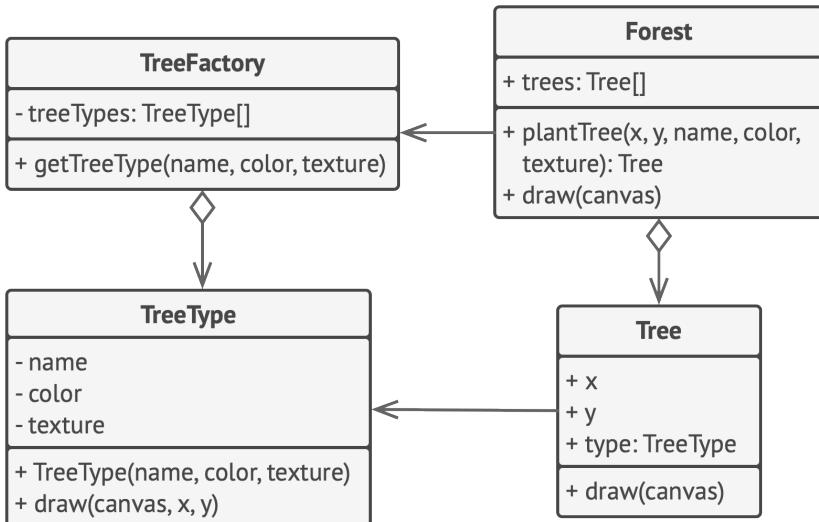
1. Ви завжди повинні пам'ятати про те, що легковаговик застосовується в програмі, яка має величезну кількість однакових об'єктів. Цих об'єктів повинно бути так багато, щоб вони не вміщалися в доступній оперативній пам'яті без додаткових хитрощів. Патерн розділяє дані цих об'єктів на дві частини – легковаговики та контексти.
2. **Легковаговик** містить стан, який повторювався в багатьох первинних об'єктах. Один і той самий легковаговик може використовуватись у зв'язці з безліччю контекстів. Стан, що

зберігається тут, називається *внутрішнім*, а той, який він отримує ззовні, – *зовнішнім*.

3. **Контекст** містить «зовнішню» частину стану, унікальну для кожного об'єкта. Контекст пов'язаний з одним з об'єктів-легковаговиків, що зберігають стан, який залишився.
4. Поведінку оригінального об'єкта найчастіше залишають у легковаговику, передаючи значення контексту через параметри методів. Тим не менше, поведінку можна розмістити й в контексті, використовуючи легковаговик як об'єкт даних.
5. **Клієнт** обчислює або зберігає контекст, тобто зовнішній стан легковаговиків. Для клієнта легковаговики виглядають як шаблонні об'єкти, які можна налаштовувати під час використання, передавши контекст через параметри.
6. **Фабрика легковаговиків** керує створенням і повторним використанням легковаговиків. Фабрика отримує запити, в яких зазначено бажаний стан легковаговика. Якщо легковаговик з таким станом вже створений, фабрика відразу його повертає, а якщо ні – створює новий об'єкт.

## # Псевдокод

У цьому прикладі **Легковаговик** допомагає заощадити операцівну пам'ять при відображення на екрані мільйонів об'єктів-дерев.



Легковаговик виділяє повторювану частину стану з основного класу `Tree` і розміщує його в додатковому класі `TreeType`. Тепер, замість зберігання повторюваних даних в усіх об'єктах, окрім дерева будуть посылатися на кілька спільних об'єктів, що зберігають ці дані. Клієнт працює з деревами через фабрику дерев, яка приховує від нього складність кешування спільних даних дерев.

Таким чином, програма буде використовувати набагато менше оперативної пам'яті, що дозволить намалювати на екрані більше дерев, використовуючи те ж саме «залізо».

```

1 // Цей клас–легковаговик містить лише частину полів, які
2 // описують дерева. На відміну, наприклад, від координат, ці
3 // поля не є унікальними для кожного дерева, оскільки декілька
4 // дерев можуть мати такий самий колір чи текстуру. Тому ми
5 // переносимо повторювані дані до одного єдиного об'єкта й
6 // посилаємося на нього з множини окремих дерев.
7 class TreeType is
8     field name
9     field color
10    field texture
11    constructor TreeType(name, color, texture) { ... }
12    method draw(canvas, x, y) is
13        // 1. Створити зображення даного типу, кольору й
14        // текстури.
15        // 2. Відобразити його на полотні в позиції X, Y.
16
17    // Фабрика легковаговиків вирішує, коли потрібно створити нового
18    // легковаговика, а коли можна обійтися існуючим.
19 class TreeFactory is
20     static field treeTypes: collection of tree types
21     static method getTreeType(name, color, texture) is
22         type = treeTypes.find(name, color, texture)
23         if (type == null)
24             type = new TreeType(name, color, texture)
25             treeTypes.add(type)
26         return type
27
28    // Контекстний об'єкт, з якого ми виділили легковаговик
29    // TreeType. У програмі можуть бути тисячі об'єктів Tree,
30    // оскільки накладні витрати на їхнє зберігання зовсім
31    // невеликі – в пам'яті треба зберігати лише три цілих числа
32    // (две координати й посилання).

```

```

33 class Tree is
34     field x,y
35     field type: TreeType
36     constructor Tree(x, y, type) { ... }
37     method draw(canvas) is
38         type.draw(canvas, this.x, this.y)
39
40 // Класи Tree і Forest є клієнтами Легковаговика. За умови, що
41 // надалі вам не потрібно розширювати клас дерев, іх можна злити
42 // докупи.
43 class Forest is
44     field trees: collection of Trees
45
46     method plantTree(x, y, name, color, texture) is
47         type = TreeFactory.getType(name, color, texture)
48         tree = new Tree(x, y, type)
49         trees.add(tree)
50
51     method draw(canvas) is
52         foreach (tree in trees) do
53             tree.draw(canvas)

```

## 💡 Застосування

- ⚡ Якщо не вистачає оперативної пам'яті для підтримки всіх потрібних об'єктів.
- ⚡ Ефективність патерна **Легковаговик** багато в чому залежить від того, як і де він використовується. Застосуйте цей патерн у випадках, коли виконано всі перераховані умови:

- у програмі використовується велика кількість об'єктів;
- через це високі витрати оперативної пам'яті;
- більшу частину стану об'єктів можна винести за межі їхніх класів;
- великі групи об'єктів можна замінити невеликою кількістю об'єктів, що розділяються, оскільки зовнішній стан винесено.

## Кроки реалізації

1. Розділіть поля класу, який стане легковаговиком, на дві частини:
  - внутрішній стан: значення цих полів однакові для великої кількості об'єктів.
  - зовнішній стан (контекст): значення полів унікальні для кожного об'єкта.
2. Залишіть поля внутрішнього стану в класі, але переконайтесь, що їхні значення неможливо змінити. Ці поля повинні ініціалізуватись тільки через конструктор.
3. Перетворіть поля зовнішнього стану на параметри методів, у яких ці поля використовувалися. Потім видаліть поля з класу.
4. Створіть фабрику, яка буде кешувати та повторно віддавати вже створені об'єкти. Клієнт повинен отримувати легковаго-

вика з певним внутрішнім станом саме з цієї фабрики, а не створювати його безпосередньо.

5. Клієнт повинен зберігати або обчислювати значення зовнішнього стану (контекст) і передавати його до методів об'єкта легковаговика.

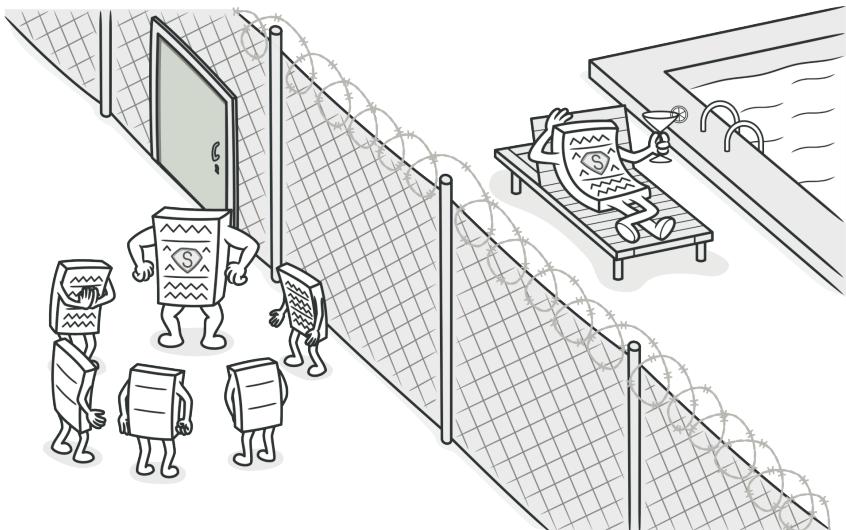
## Переваги та недоліки

- ✓ Заощаджує оперативну пам'ять.
- ✗ Витрачає процесорний час на пошук/обчислення контексту.
- ✗ Ускладнює код програми внаслідок введення безлічі додаткових класів.

## Відносини з іншими патернами

- Компонувальник часто поєднують з Легковаговиком, щоб реалізувати спільні гілки дерева та заощадити при цьому пам'ять.
- Легковаговик показує, як створювати багато дрібних об'єктів, а Фасад показує, як створити один об'єкт, який відображає цілу підсистему.
- Патерн Легковаговик може нагадувати Однака, якщо для конкретного завдання ви змогли зменшити кількість об'єктів до одного. Але пам'ятайте, що між патернами є дві суттєві відмінності:

1. На відміну від *Одинака*, ви можете мати безліч об'єктів-легковаговиків.
2. Об'єкти-легковаговики повинні бути незмінними, тоді як об'єкт-одинак допускає зміну свого стану.



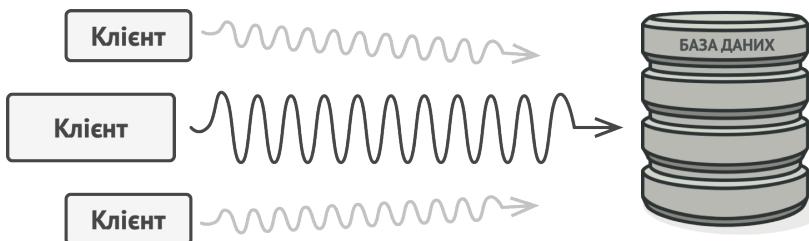
# ЗАМІСНИК

Також відомий як: *Proxy*

**Замісник** – це структурний патерн проєктування, що дає змогу підставляти замість реальних об'єктів спеціальні об'єкти-замінники. Ці об'єкти перехоплюють виклики до оригінального об'єкта, дозволяючи зробити щось до чи після передачі виклику оригіналові.

## :( Проблема

Для чого взагалі контролювати доступ до об'єктів? Розглянемо такий приклад: у вас є зовнішній ресурсоємний об'єкт, який потрібен не весь час, а лише зрідка.



*Запити до бази даних можуть бути дуже повільними.*

Ми могли б створювати цей об'єкт не на самому початку програми, а тільки тоді, коли він реально кому-небудь знадобиться. Кожен клієнт об'єкта отримав би деякий код відкладеної ініціалізації. Це, ймовірно, привело б до дублювання великої кількості коду.

В ідеалі цей код хотілося б помістити безпосередньо до службового класу, але це не завжди можливо. Наприклад, код класу може знаходитися в закритій сторонній бібліотеці.

## :)) Рішення

Патерн Замісник пропонує створити новий клас-дублер, який має той самий інтерфейс, що й оригінальний службовий об'єкт. При отриманні запиту від клієнта об'єкт-замісник

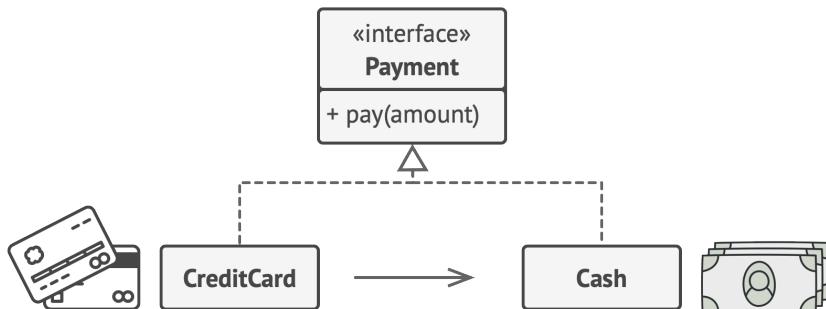
сам би створював примірник службового об'єкта та переадресовував би йому всю реальну роботу.



*Замісник «прикладається» базою даних, прискорюючи роботу внаслідок ледачої ініціалізації і кешування запитів, що повторюються.*

Але в чому ж його користь? Ви могли б помістити до класу замісника якусь проміжну логіку, що виконувалася б до або після викликів цих самих методів чинного об'єкта. А завдяки однаковому інтерфейсу об'єкт-замісник можна передати до будь-якого коду, що очікує на сервісний об'єкт.

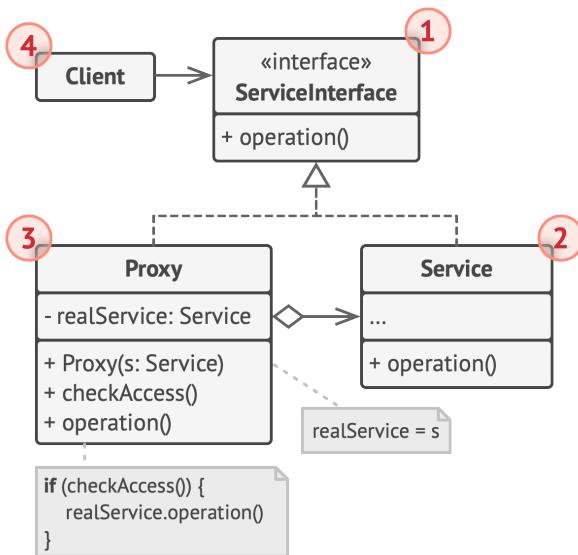
## 🚗 Аналогія з життя



*Платіжною карткою можна розраховуватися так само, як і готівкою.*

Платіжна картка – це замісник пачки готівки. І чек, і готівка мають спільний інтерфейс – ними обома можна оплачувати товари. Вигода покупця в тому, що не потрібно носити з собою «тонні» готівки. З іншого боку власник магазину не змушений замовляти клопітку інкасацію коштів з магазину, бо вони потрапляють безпосередньо на його банківський рахунок.

## Структура

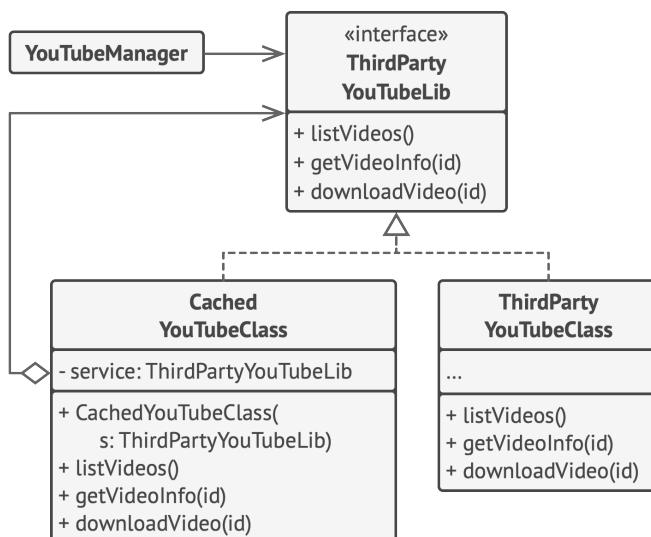


1. **Інтерфейс сервісу** визначає загальний інтерфейс для сервісу й замісника. Завдяки цьому об'єкт замісника можна використовувати там, де очікується об'єкт сервісу.
2. **Сервіс** містить корисну бізнес-логіку.

3. **Замісник** зберігає посилання на об'єкт сервісу. Після того, як замісник закінчує свою роботу (наприклад, ініціалізацію, логування, захист або інше), він передає виклики вкладеному сервісу. Замісник може сам відповідати за створення й видалення об'єкта сервісу.
4. **Клієнт** працює з об'єктами через інтерфейс сервісу. Завдяки цьому його можна «обдурити», підмінивши об'єкт сервісу об'єктом замісника.

## # Псевдокод

У цьому прикладі **Замісник** допомагає додати до програми механізм ледачої ініціалізації та кешування результатів роботи бібліотеки інтеграції з YouTube.



Приклад кешування результатів роботи реального сервісу за допомогою замісника.

Оригінальний об'єкт починав завантаження з мережі, навіть якщо користувач повторно запитував одне й те саме відео. Замісник завантажує відео тільки один раз, використовуючи для цього службовий об'єкт, але в інших випадках повертає закешований файл.

```
1 // Інтерфейс віддаленого сервісу.  
2 interface ThirdPartyYouTubeLib is  
3     method listVideos()  
4     method getVideoInfo(id)  
5     method downloadVideo(id)  
6  
7 // Конкретна реалізація сервісу. Методи цього класу запитують у  
8 // YouTube різну інформацію. Швидкість запиту залежить не лише  
9 // від якості інтернет-каналу користувача, але й від стану  
10 // самого YouTube. Тому, чим більше буде викликів до сервісу,  
11 // тим менш відзвинною стане програма.  
12 class ThirdPartyYouTubeClass implements ThirdPartyYouTubeLib is  
13     method listVideos() is  
14         // Отримати список відеороликів за допомогою API  
15         // YouTube.  
16  
17     method getVideoInfo(id) is  
18         // Отримати детальну інформацію про якийсь відеоролик.  
19  
20     method downloadVideo(id) is  
21         // Завантажити відео з YouTube.  
22  
23     // З іншого боку, можна кешувати запити до YouTube і не  
24     // повторювати їх деякий час, доки кеш не застаріє. Але внести  
25     // цей код безпосередньо в сервісний клас неможливо, бо він
```

```
26 // знаходиться у сторонній бібліотеці. Тому ми помістимо логіку
27 // кешування в окремий клас—обгортку. Він буде делегувати запити
28 // сервісному об'єкту, тільки якщо потрібно безпосередньо
29 // відіслати запит.
30 class CachedYouTubeClass implements ThirdPartyYouTubeLib is
31     private field service: ThirdPartyYouTubeLib
32     private field listCache, videoCache
33     field needReset
34
35     constructor CachedYouTubeClass(service: ThirdPartyYouTubeLib) is
36         this.service = service
37
38     method listVideos() is
39         if (listCache == null || needReset)
40             listCache = service.listVideos()
41         return listCache
42
43     method getVideoInfo(id) is
44         if (videoCache == null || needReset)
45             videoCache = service.getVideoInfo(id)
46         return videoCache
47
48     method downloadVideo(id) is
49         if (!downloadExists(id) || needReset)
50             service.downloadVideo(id)
51
52 // Клас GUI, який використовує сервісний об'єкт. Замість
53 // реального сервісу, ми підсунемо йому об'єкт-замісник. Клієнт
54 // нічого не помітить, так як замісник має той самий інтерфейс,
55 // що й сервіс.
56 class YouTubeManager is
57     protected field service: ThirdPartyYouTubeLib
```

```

58 constructor YouTubeManager(service: ThirdPartyYouTubeLib) is
59     this.service = service
60
61 method renderVideoPage(id) is
62     info = service.getVideoInfo(id)
63     // Відобразити сторінку відеоролика.
64
65 method renderListPanel() is
66     list = service.listVideos()
67     // Відобразити список прев'ю відеороликів.
68
69 method reactOnUserInput() is
70     renderVideoPage()
71     renderListPanel()
72
73 // Конфігураційна частина програми створює та передає клієнтам
74 // об'єкт замісника.
75 class Application is
76     method init() is
77         YouTubeService = new ThirdPartyYouTubeClass()
78         YouTubeProxy = new CachedYouTubeClass(YouTubeService)
79         manager = new YouTubeManager(YouTubeProxy)
80         manager.reactOnUserInput()

```

## Застосування

 **Лінива ініціалізація (віртуальний проксі).** Коли у вас є важкий об'єкт, який завантажує дані з файлової системи або бази даних.

- ⚡ Замість того, щоб завантажувати дані відразу після старту програми, можна заощадити ресурси й створити об'єкт тоді, коли він дійсно знадобиться.
  
- ⚡ **Захист доступу (захищаючий проксі).** Коли в програмі є різні типи користувачів, і вам хочеться захистити об'єкт від неавторизованого доступу. Наприклад, якщо ваші об'єкти – це важлива частина операційної системи, а користувачі – сторонні програми (корисні чи шкідливі).
  
- ⚡ Проксі може перевіряти доступ під час кожного виклику та передавати виконання службовому об'єкту, якщо доступ дозволено.
  
- ⚡ **Локальний запуск сервісу (віддалений проксі).** Коли спріважній сервісний об'єкт знаходиться на віддаленому сервері.
  
- ⚡ У цьому випадку замісник транслює запити клієнта у виклики через мережу по протоколу, який є зрозумілим віддаленому сервісу.
  
- ⚡ **Логування запитів (логуючий проксі).** Коли потрібно зберігати історію звернень до сервісного об'єкта.
  
- ⚡ Замісник може зберігати історію звернення клієнта до сервісного об'єкта.

 **Кешування об'єктів («розумне» посилання).** Коли потрібно кешувати результати запитів клієнтів і керувати їхнім життєвим циклом.

 Замісник може підраховувати кількість посилань на сервісний об'єкт, які були віддані клієнту та залишаються активними. Коли всі посилання звільняться, можна буде звільнити і сам сервісний об'єкт (наприклад, закрити підключення до бази даних).

Крім того, Замісник може відстежувати, чи клієнт не змінював сервісний об'єкт. Це дозволить повторно використовувати об'єкти й суттєво заощаджувати ресурси, особливо якщо мова йде про великі «ненажерливі» сервіси.

## Кроки реалізації

1. Визначте інтерфейс, який би зробив замісника та оригінальний об'єкт взаємозамінними.
2. Створіть клас замісника. Він повинен містити посилання на сервісний об'єкт. Частіше за все сервісний об'єкт створюється самим замісником. У рідкісних випадках замісник отримує готовий сервісний об'єкт від клієнта через конструктор.
3. Реалізуйте методи замісника в залежності від його призначення. У більшості випадків, виконавши якусь корисну роботу, методи замісника повинні передати запит сервісному об'єкту.

4. Подумайте про введення фабрики, яка б вирішувала, який з об'єктів створювати: замісника або реальний сервісний об'єкт. Проте, з іншого боку, ця логіка може бути вкладена до створюючого методу самого замісника.
5. Подумайте, чи не реалізувати вам лінівну ініціалізацію сервісного об'єкта при першому зверненні клієнта до методів замісника.

## ΔΔ Переваги та недоліки

- ✓ Дозволяє контролювати сервісний об'єкт непомітно для клієнта.
- ✓ Може працювати, навіть якщо сервісний об'єкт ще не створено.
- ✓ Може контролювати життєвий цикл службового об'єкта.
- ✗ Ускладнює код програми внаслідок введення додаткових класів.
- ✗ Збільшує час отримання відклику від сервісу.

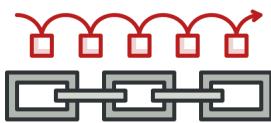
## ↔ Відносини з іншими патернами

- Адаптер надає класу альтернативний інтерфейс. Декоратор надає розширеній інтерфейс. Замісник надає той самий інтерфейс.

- **Фасад** схожий на **Замісник** тим, що замінює складну підсистему та може сам її ініціалізувати. Але, на відміну від **Фасаду**, **Замісник** має такий самий інтерфейс, що і його службовий об'єкт, завдяки чому їх можна взаємозамінити.
- **Декоратор** та **Замісник** мають схожі структури, але різні призначення. Вони схожі тим, що обидва побудовані на композиції та делегуванні роботи іншому об'єкту. Патерни відрізняються тим, що **Замісник** сам керує життям сервісного об'єкта, а обгортання **Декораторів** контролюється клієнтом.

# Поведінкові патерни

Ці патерни вирішують завдання ефективної та безпечної взаємодії між об'єктами програми.



## Ланцюжок обов'язків

Chain of Responsibility

Дає змогу передавати запити послідовно ланцюжком обробників. Кожен наступний обробник вирішує, чи може він обробити запит сам і чи варто передавати запит далі ланцюжком.



## Команда

Command

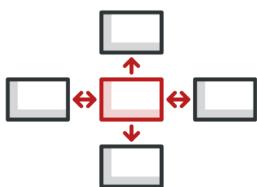
Перетворює запити на об'єкти, дозволяючи передавати їх як аргументи під час виклику методів, ставити запити в чергу, логувати їх, а також підтримувати скасування операцій.



## Ітератор

Iterator

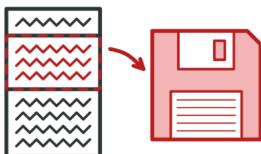
Дає змогу послідовно обходити елементи складових об'єктів, не розкриваючи їхньої внутрішньої організації.



## Посередник

Mediator

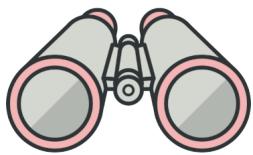
Дає змогу зменшити зв'язаність великої кількості класів між собою, завдяки переміщенню цих зв'язків до одного класу-посередника.



## Знімок

Memento

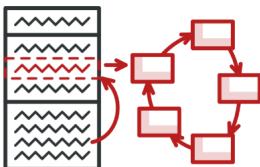
Дає змогу зберігати та відновлювати минулий стан об'єктів, не розкриваючи подробиць їхньої реалізації.



## Спостерігач

Observer

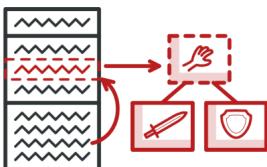
Створює механізм підписки, що дає змогу одним об'єктам стежити й реагувати на події, які відбуваються в інших об'єктах.



## Стан

State

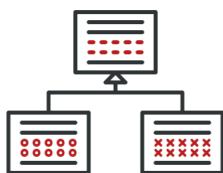
Дає змогу об'єктам змінювати поведінку в залежності від їхнього стану. Ззовні створюється враження, ніби змінився клас об'єкта.



## Стратегія

Strategy

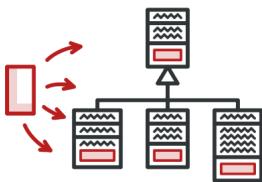
Визначає сімейство схожих алгоритмів і розміщує кожен з них у власному класі. Після цього алгоритми можна замінити один на інший прямо під час виконання програми.



## Шаблонний метод

Template Method

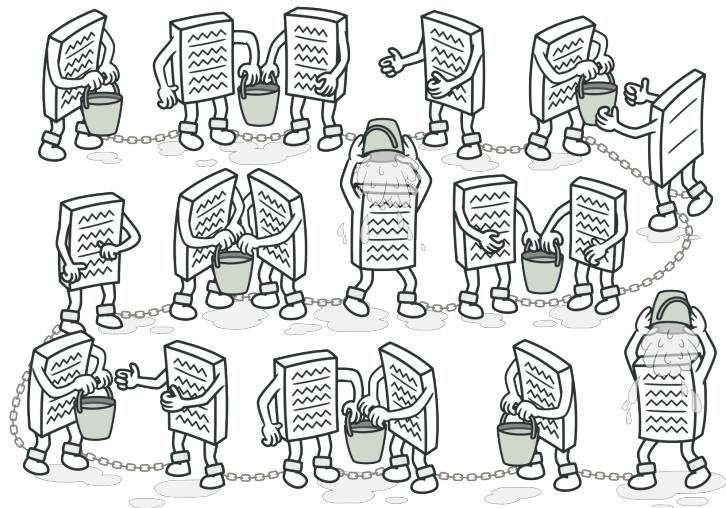
Визначає кістяк алгоритму, перекладаючи відповіальність за деякі його кроки на підкласи. Патерн дозволяє підкласам перевизначати кроки алгоритму, не змінюючи його загальної структури.



## Відвідувач

Visitor

Дає змогу додавати до програми нові операції, не змінюючи класи об'єктів, над якими ці операції можуть виконуватися.



# ЛАНЦЮЖОК ОБОВ'ЯЗКІВ

Також відомий як: Ланцюг відповідальностей, CoR, *Chain of Command, Chain of Responsibility*

**Ланцюжок обов'язків** – це поведінковий патерн проектування, що дає змогу передавати запити послідовно ланцюжком обробників. Кожен наступний обробник вирішує, чи може він обробити запит сам і чи варто передавати запит далі ланцюжком.

## (:() Проблема

Уявіть, що ви робите систему прийому онлайн-замовлень. Ви хочете обмежити до неї доступ так, щоб тільки авторизовані користувачі могли створювати замовлення. Крім того, певні користувачі, які володіють правами адміністратора, повинні мати повний доступ до замовлень.

Ви швидко збагнули, що ці перевірки потрібно виконувати послідовно. Адже користувача можна спробувати «залигувати» у систему, якщо його запит містить логін і пароль. Але, якщо така спроба не вдалась, то перевіряти розширені права доступу просто немає сенсу.



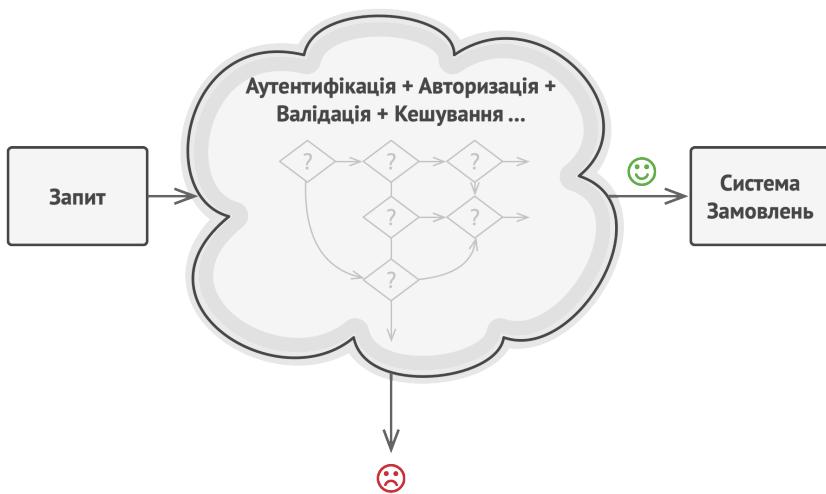
*Запит проходить ряд перевірок перед доступом до системи замовлень.*

Протягом наступних кількох місяців вам довелося додати ще декілька таких послідовних перевірок.

- Хтось слушно зауважив, що непогано було б перевіряти дані, що передаються в запиті, перед тим, як вносити їх до систе-

ми — раптом запит містить дані про покупку неіснуючих продуктів.

- Хтось запропонував блокувати масові надсилання форми з одним і тим самим логіном, щоб запобігти підбору паролів ботами.
- Хтось зауважив, що непогано було б діставати форму замовлення з кешу, якщо вона вже була одного разу показана.



*З часом код перевірок стає все більш заплутаним.*

З кожною новою «фічою» код перевірок, що виглядав як величезний клубок умовних операторів, все більше і більше «розбухав». При зміні одного правила доводилося змінювати код усіх інших перевірок. А щоб застосувати перевірки до інших ресурсів, довелося також продублювати їхній код в інших класах.

Підтримувати такий код стало не тільки вкрай незручно, але й витратно. Аж ось одного прекрасного дня ви отримуєте завдання рефакторингу...

## Рішення

Як і багато інших поведінкових патернів, ланцюжок обов'язків базується на тому, щоб перетворити окремі поведінки на об'єкти. У нашому випадку кожна перевірка переїде до окремого класу з одним методом виконання. Дані запиту, що перевіряється, передаватимуться до методу як аргументи.

А тепер справді важливий етап. Патерн пропонує зв'язати всі об'єкти обробників в один ланцюжок. Кожен обробник міститиме посилання на наступного обробника в ланцюзі. Таким чином, після отримання запиту обробник зможе не тільки опрацювати його самостійно, але й передати обробку наступному об'єкту в ланцюжку.

Передаючи запити до першого обробника ланцюжка, ви можете бути впевнені, що всі об'єкти в ланцюзі зможуть його обробити. При цьому довжина ланцюжка не має жодного значення.

І останній штрих. Обробник не обов'язково повинен передавати запит далі. Причому ця особливість може бути використана різними шляхами.

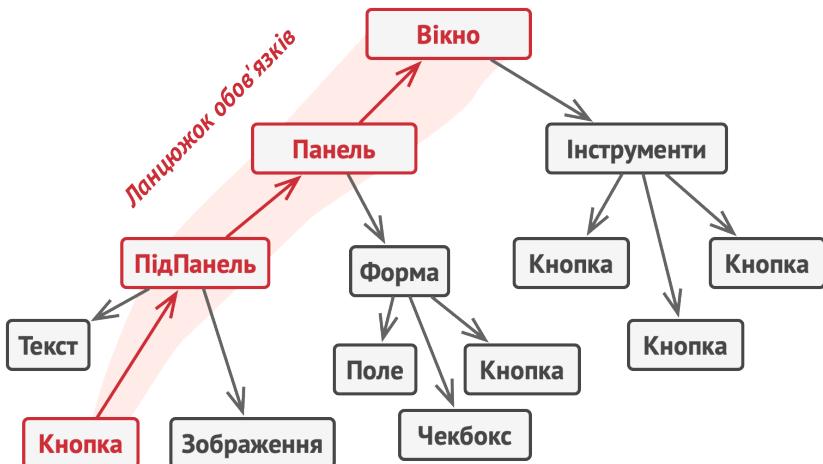
У прикладі з фільтрацією доступу обробники переривають подальші перевірки, якщо поточну перевірку не пройдено. Адже немає сенсу витрачати даремно ресурси, якщо і так зрозуміло, що із запитом щось не так.



*Обробники слідують в ланцюжку один за іншим.*

Але є й інший підхід, коли обробники переривають ланцюг, тільки якщо вони можуть обробити запит. У цьому випадку запит рухається ланцюгом, поки не знайдеться обробник, який зможе його обробити. Дуже часто такий підхід використовується для передачі подій, що генеруються у класах графічного інтерфейсу внаслідок взаємодії з користувачем.

Наприклад, коли користувач клікає по кнопці, програма будує ланцюжок з об'єкта цієї кнопки, всіх її батьківських елементів і загального вікна програми на кінці. Подія кліку передається цим ланцюжком до тих пір, поки не знайдеться об'єкт, здатний її обробити. Цей приклад примітний ще й тим, що ланцюжок завжди можна виділити з деревоподібної структури об'єктів, в яку зазвичай і згорнуті елементи користувачького інтерфейсу.



Ланцюжок можна виділити навіть із дерева об'єктів.

Дуже важливо, щоб усі об'єкти ланцюжка мали спільний інтерфейс. Зазвичай кожному конкретному обробникові достатньо знати тільки те, що наступний об'єкт ланцюжка має метод `виконати`. Завдяки цьому зв'язки між об'єктами ланцюжка будуть більш гнучкими. Крім того, ви зможете формувати ланцюжки на льоту з різноманітних об'єктів, не прив'язуючись до конкретних класів.

## Аналогія з життя

Ви купили нову відеокарту. Вона автоматично визначилася і почала працювати під Windows, але у вашій улюблений Ubuntu «завести» її не вдалося. Ви телефонуєте до служби підтримки виробника, але без особливих сподівань на вирішення проблеми.

Спочатку ви чуєте голос автовідповідача, який пропонує вибір з десяти стандартних рішень. Жоден з варіантів не підходить, і робот з'єднує вас з живим оператором.

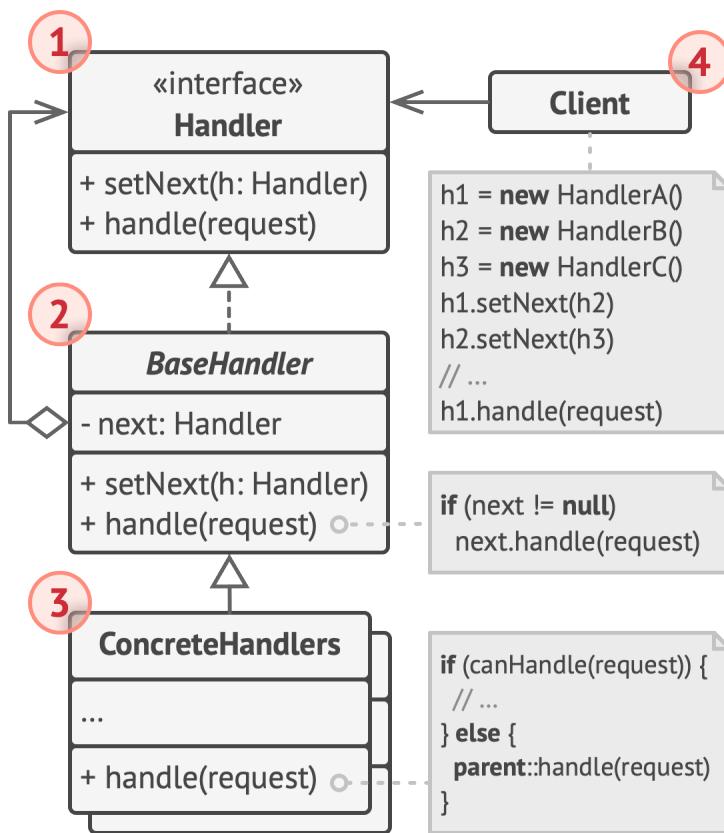


Приклад спілкування з підтримкою.

На жаль, звичайний оператор підтримки вміє спілкуватися тільки завченими фразами і давати тільки шаблонні відповіді. Після чергової пропозиції «вимкнути і ввімкнути комп’ютер» ви просите зв’язати вас зі справжніми інженерами.

Оператор перекидає дзвінок черговому інженерові, який знемагає від нудьги у своїй комірчині. От він вже точно знає, як вам допомогти! Інженер розповідає вам, де завантажити драйвери та як налаштувати їх під Ubuntu. Запит вирішено. Ви кладете слухавку.

## Структура



1. **Обробник** визначає спільний для всіх конкретних обробників інтерфейс. Зазвичай достатньо описати один метод обробки запитів, але іноді тут може бути оголошений і метод встановлення наступного обробника.
2. **Базовий обробник** – опціональний клас, який дає змогу позбутися дублювання одного і того самого коду в усіх конкретних обробниках.

Зазвичай цей клас має поле для зберігання посилання на наступного обробника у ланцюжку. Клієнт зв'язує обробників у ланцюг, подаючи посилання на наступного обробника через конструктор або сетер поля. Також в цьому класі можна реалізувати базовий метод обробки, який би просто перенаправляв запити наступному обробнику, перевіривши його наявність.

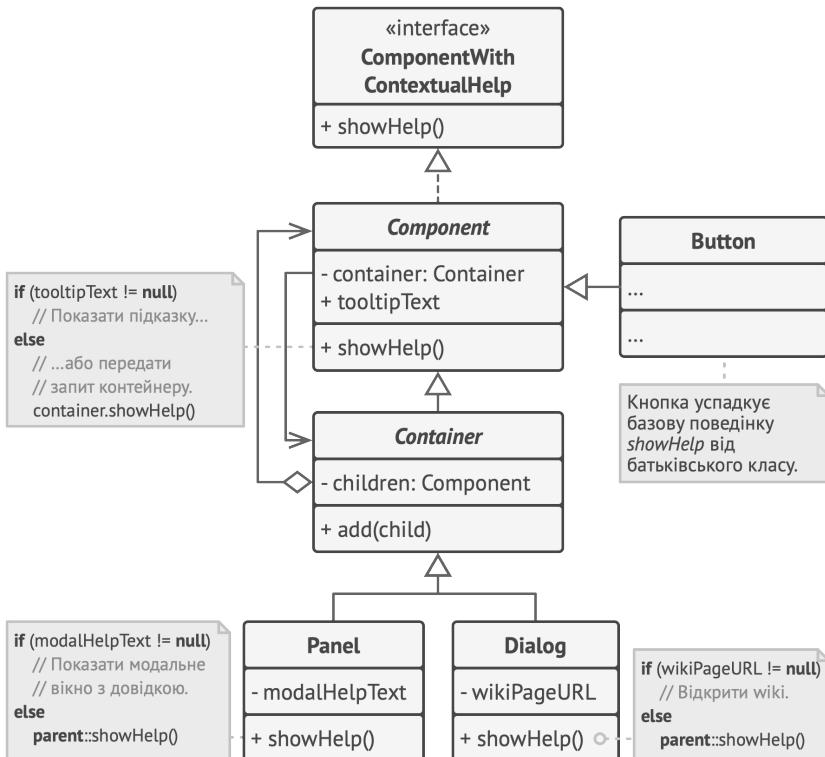
3. **Конкретні обробники** містять код обробки запитів. При отриманні запиту кожен обробник вирішує, чи може він обробити запит, а також чи варто передати його наступному об'єкту.

У більшості випадків обробники можуть працювати самостійно і бути незмінними, отримавши всі необхідні деталі через параметри конструктора.

4. **Клієнт** може сформувати ланцюжок лише один раз і використовувати його протягом всього часу роботи програми, так і перебудовувати його динамічно, залежно від логіки програми. Клієнт може відправляти запити будь-якому об'єкту ланцюжка, не обов'язково першому з них.

## # Псевдокод

У цьому прикладі **Ланцюжок обов'язків** відповідає за показ контекстної допомоги для активних елементів інтерфейсу користувача.

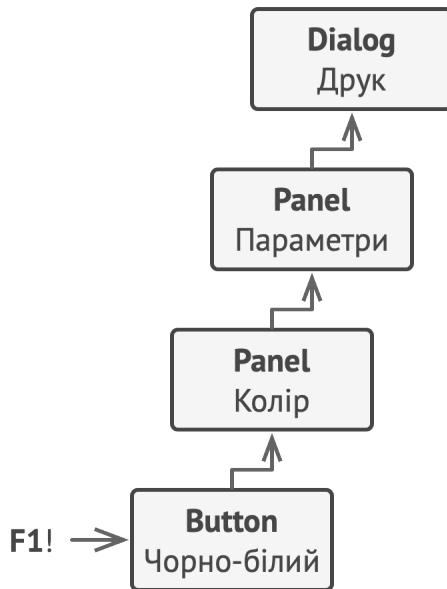


Графічний інтерфейс побудований за допомогою компонувальника, де кожен елемент має посилання на свій елемент-контейнер. Ланцюжок можна вибудувати, пройшовши по всіх контейнерах, у які вкладено елемент.

Графічний інтерфейс програми зазвичай структурований у вигляді дерева. Клас **Діалог**, який відображає все вікно програми, — це корінь дерева. Діалог містить **Панелі**, які, в свою чергу, можуть містити або інші вкладені панелі, або прості елементи на зразок **Кнопок**.

Прості елементи можуть показувати невеликі підказки, якщо для них вказано допоміжний текст. Але є й більш складні

компоненти, для яких цей спосіб демонстрації допомоги занадто простий. Вони визначають власний спосіб відображення контекстної допомоги.



*Приклад виклику контекстної допомоги у ланцюжку об'єктів UI.*

Коли користувач наводить вказівник миші на елемент і тисне клавішу **F1**, програма надсилає цьому елементу запит щодо показу допомоги. Якщо він не містить жодної довідкової інформації, запит подорожує списком контейнерів елемента, доки не знаходиться той, що може відобразити допомогу.

```
1 // Інтерфейс обробників.
2 interface ComponentWithContextualHelp is
3     method showHelp()
4
5
6 // Базовий клас простих компонентів.
7 abstract class Component implements ComponentWithContextualHelp is
8     field tooltipText: string
9
10    // Контейнер, що містить компонент, служить в якості
11    // наступної ланки ланцюга.
12    protected field container: Container
13
14    // Базова поведінка компонента заключається в тому, щоб
15    // показати вспливаючу підказку, якщо для неї задано текст.
16    // А якщо ні – перенаправити запит своєму контейнеру, якщо
17    // той існує.
18    method showHelp() is
19        if (tooltipText != null)
20            // Показати підказку.
21        else
22            container.showHelp()
23
24
25    // Контейнери можуть містити як прості компоненти, так й інші
26    // контейнери. Тут формуються зв'язки ланцюжка. Клас успадкує
27    // метод showHelp від свого батька.
28 abstract class Container extends Component is
29     protected field children: array of Component
30     method add(child) is
31         children.add(child)
32         child.container = this
```

```

33 // Більшість конкретних компонентів влаштує базова поведінка
34 // допомоги із вспливаючою підказкою, що вони успадкують від
35 // класу Component.
36 class Button extends Component is
37     ...
38
39 // Але складні компоненти можуть перевизначати метод показу
40 // допомоги по-своєму. Але і в цьому випадку вони завжди можуть
41 // повернутися до базової реалізації, викликавши метод батька.
42 class Panel extends Container is
43     field modalHelpText: string
44
45     method showHelp() is
46         if (modalHelpText != null)
47             // Показати модальне вікно з допомогою.
48         else
49             super.showHelp()
50
51 // ...те саме, що й вище...
52 class Dialog extends Container is
53     field wikiPageURL: string
54
55     method showHelp() is
56         if (wikiPageURL != null)
57             // Відкрити сторінку Wiki в браузері.
58         else
59             super.showHelp()
60
61
62 // Клієнтський код.
63 class Application is
64     // Кожна програма конфігурує ланцюжок по-своєму.

```

```

65 method createUI() is
66     dialog = new Dialog("Budget Reports")
67     dialog.wikiPageURL = "http://..."
68     panel = new Panel(0, 0, 400, 800)
69     panel.modalHelpText = "This panel does..."
70     ok = new Button(250, 760, 50, 20, "OK")
71     ok.tooltipText = "This is an OK button that..."
72     cancel = new Button(320, 760, 50, 20, "Cancel")
73     // ...
74     panel.add(ok)
75     panel.add(cancel)
76     dialog.add(panel)
77
78 // Уявіть, що тут відбудеться.
79 method onF1KeyPress() is
80     component = this.getComponentAtMouseCoords()
81     component.showHelp()

```

## Застосування

-  Якщо програма має обробляти різноманітні запити багатьма способами, але заздалегідь невідомо, які конкретно запити надходитимуть і які обробники для них знадобляться.
-  За допомогою Ланцюжка обов'язків ви можете зв'язати потенційних обробників в один ланцюг і по отриманню запита по черзі питати кожного з них, чи не хоче він обробити даний запит.

-  **Якщо важливо, щоб обробники виконувалися один за іншим у суворому порядку.**
-  Ланцюжок обов'язків дозволяє запускати обробників один за одним у тій послідовності, в якій вони стоять в ланцюзі.
-  **Якщо набір об'єктів, здатних обробити запит, повинен задаватися динамічно.**
-  У будь-який момент ви можете втрутитися в існуючий ланцюжок і перевизначити зв'язки так, щоби прибрati або додати нову ланку.

## Кроки реалізації

- Створіть інтерфейс обробника і описіть в ньому основний метод обробки.

Продумайте, в якому вигляді клієнт повинен передавати дані запиту до обробника. Найгнучкіший спосіб – це перетворити дані запиту на об'єкт і повністю передавати його через параметри методу обробника.

- Є сенс у тому, щоб створити абстрактний базовий клас обробників, аби не дублювати реалізацію методу отримання наступного обробника в усіх конкретних обробниках.

Додайте до базового обробника поле для збереження посилання на наступний елемент ланцюжка. Встановлюйте початкове значення цього поля через конструктор. Це зробить

об'єкти обробників незмінюваними. Але якщо програма передбачає динамічну перебудову ланцюжків, можете додати ще й сетер для поля.

Реалізуйте базовий метод обробки так, щоб він перенаправляв запит наступному об'єкту, перевіривши його наявність. Це дозволить повністю приховати поле-посилання від підкласів, давши їм можливість передавати запити далі ланцюгом, звертаючись до батьківської реалізації методу.

3. Один за іншим створіть класи конкретних обробників та реалізуйте в них методи обробки запитів. При отриманні запиту кожен обробник повинен вирішити:
  - Чи може він обробити запит, чи ні?
  - Чи потрібно передавати запит наступному обробникові, чи ні?
4. Клієнт може збирати ланцюжок обробників самостійно, спираючись на свою бізнес-логіку, або отримувати вже готові ланцюжки ззовні. В останньому випадку ланцюжки збираються фабричними об'єктами, спираючись на конфігурацію програми або параметри оточення.
5. Клієнт може надсилати запити будь-якому обробникові ланцюга, а не лише першому. Запит передаватиметься ланцюжком, допоки який-небудь обробник не відмовиться передавати його далі або коли буде досягнуто кінець ланцюга.
6. Клієнт повинен знати про динамічну природу ланцюжка і бути готовим до таких випадків:

- Ланцюжок може складатися з одного об'єкта.
- Запити можуть не досягати кінця ланцюга.
- Запити можуть досягати кінця, залишаючись необробленими.

## **ΔΦ Переваги та недоліки**

- ✓ Зменшує залежність між клієнтом та обробниками.
- ✓ Реалізує *принцип єдиного обов'язку*.
- ✓ Реалізує *принцип відкритості/закритості*.
- ✗ Запит може залишитися ніким не опрацьованим.

## **↔ Відносини з іншими патернами**

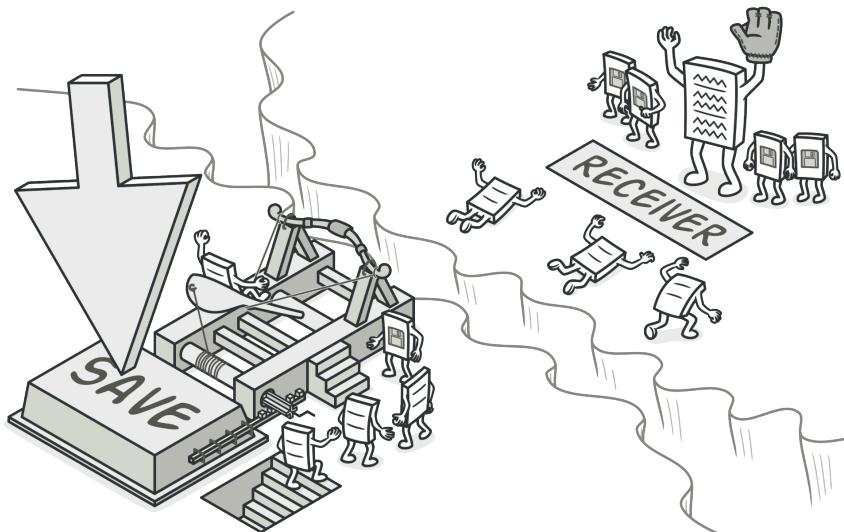
- Ланцюжок обов'язків, Команда Посередник та Спостерігач показують різні способи роботи тих, хто надсилає запити, та тих, хто їх отримує:
  - *Ланцюжок обов'язків* передає запит послідовно через ланцюжок потенційних отримувачів, очікуючи, що один з них обробить запит.
  - *Команда* встановлює непрямий односторонній зв'язок від відправників до одержувачів.
  - *Посередник* прибирає прямий зв'язок між відправниками та одержувачами, змушуючи їх спілкуватися опосередковано, через себе.

- Спостерігач передає запит одночасно всім зацікавленим одержувачам, але дозволяє їм динамічно підписуватися або відписуватися від таких повідомлень.
- **Ланцюжок обов'язків** часто використовують разом з **Компонувальником**. У цьому випадку запит передається від дочірніх компонентів до їхніх батьків.
- Обробники в **Ланцюжкові обов'язків** можуть бути виконані у вигляді **Команд**. В цьому випадку роль запиту відіграє контекст команд, який послідовно подається доожної команди у ланцюгу.

Але є й інший підхід, в якому сам запит є *Командою*, надісланою ланцюжком об'єктів. У цьому випадку одна і та сама операція може бути застосована до багатьох різних контекстів, представлених у вигляді ланцюжка.

- **Ланцюжок обов'язків** та **Декоратор** мають дуже схожі структури. Обидва патерни базуються на принципі рекурсивного виконання операції через серію пов'язаних об'єктів. Але є декілька важливих відмінностей.

Обробники в *Ланцюжку обов'язків* можуть виконувати довільні дії, незалежні одна від одної, а також у будь-який момент переривати подальшу передачу ланцюжком. З іншого боку, *Декоратори* розширяють певну дію, не ламаючи інтерфейс базової операції і не перериваючи виконання інших декораторів.



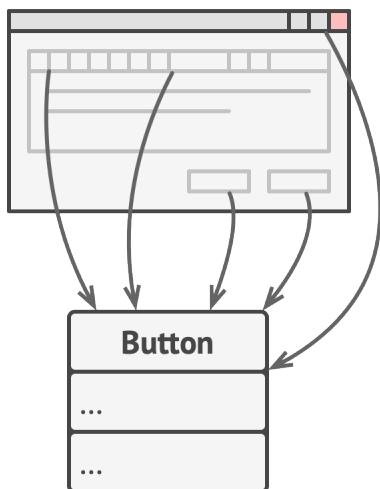
# КОМАНДА

Також відомий як: *Дія, Транзакція, Action, Command*

**Команда** – це поведінковий патерн проектування, який перетворює запити на об'єкти, дозволяючи передавати їх як аргументи під час виклику методів, ставити запити в чергу, логувати їх, а також підтримувати скасування операцій.

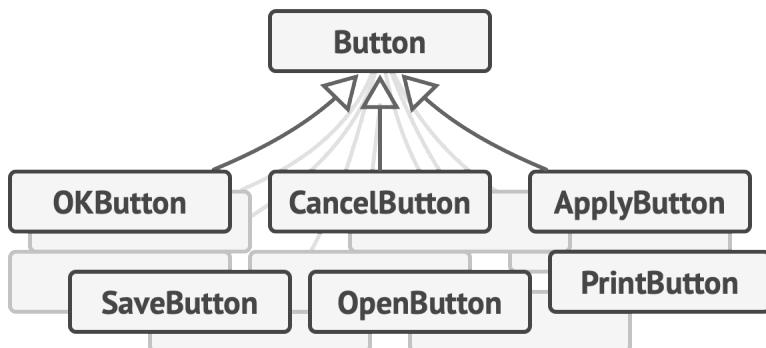
## :( Проблема

Уявіть, що ви працюєте над програмою текстового редактора. Якраз підійшов час розробки панелі керування. Ви створили клас гарних Кнопок і хочете використовувати його для всіх кнопок програми, починаючи з панелі керування та закінчуючи звичайними кнопками в діалогах.



*Всі кнопки програми успадковані від одного класу.*

Усі ці кнопки, хоч і виглядають схоже, але виконують різні команди. Виникає запитання: куди розмістити код обробників кліків по цих кнопках? Найпростіше рішення – це створити підкласи для кожної кнопки та перевизначити в них методи дії для різних завдань.



Але скоро стало зрозуміло, що такий підхід нікуди не годиться. По-перше, з'являється дуже багато підкласів. По-друге, код кнопок, який відноситься до графічного інтерфейсу, починає залежати від класів бізнес-логіки, яка досить часто змінюється.



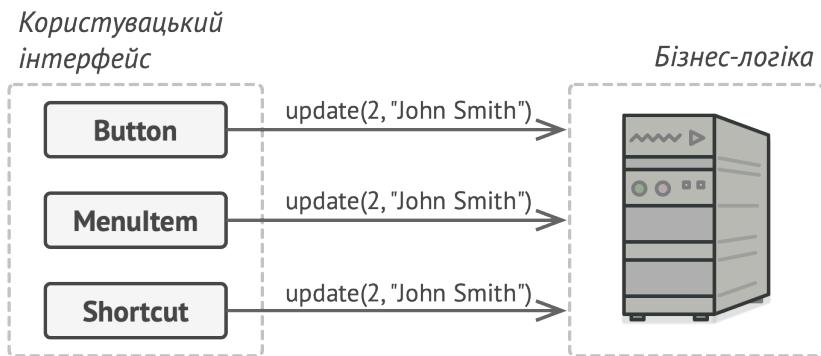
Кілька класів дублюють одну і ту саму функціональність.

Проте, найгірше ще попереду, адже деякі операції, на кшталт «зберегти», можна викликати з декількох місць: натиснувши кнопку на панелі керування, викликавши контекстне меню або натиснувши клавіші `Ctrl+S`. Коли в програмі були тільки кнопки, код збереження був тільки у підкласі `SaveButton`. Але тепер його доведеться продублювати ще в два класи.

## 😊 Рішення

Хороші програми зазвичай структурують у вигляді шарів. Найпоширеніший приклад – це шари користувачького інтерфейсу та бізнес-логіки. Перший лише має гарне зображення для користувача, але коли потрібно зробити щось важливе, інтерфейс користувача «просить» шар бізнес-логіки зайнятися цим.

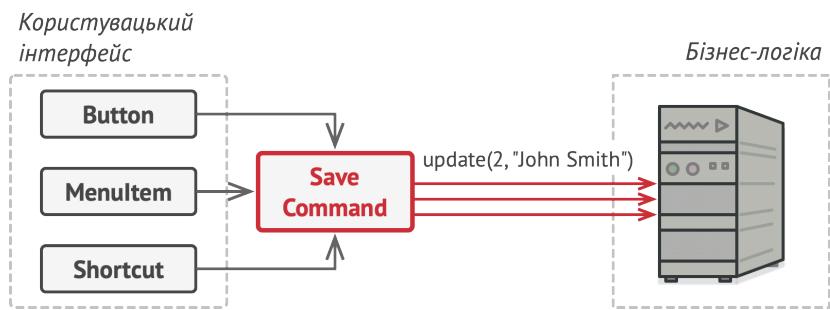
У дійсності це виглядає так: один з об'єктів інтерфейсу користувача викликає метод одного з об'єктів бізнес-логіки, передаючи до нього якісь параметри.



*Прямий доступ з UI до бізнес-логіки.*

Патерн Команда пропонує більше не надсилати такі виклики безпосередньо. Замість цього кожен виклик, що відрізняється від інших, слід звернути у власний клас з єдиним методом, який і здійснюватиме виклик. Такий звуться *командою*.

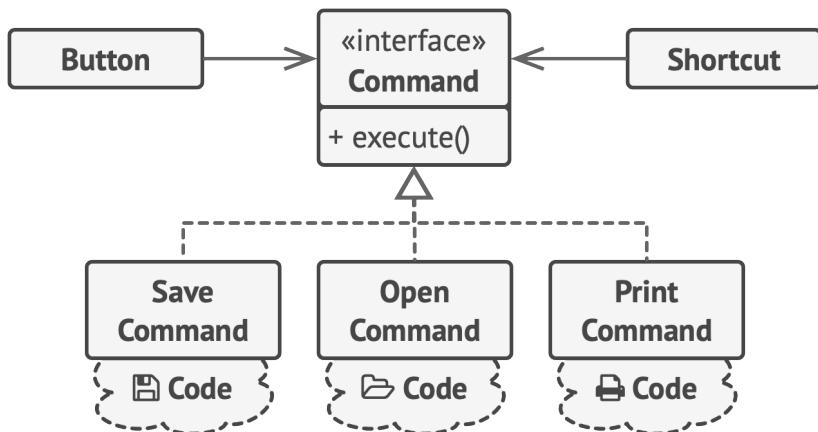
До об'єкта інтерфейсу можна буде прив'язати об'єкт команди, який знає, кому і в якому вигляді слід відправляти запити. Коли об'єкт інтерфейсу буде готовий передати запит, він виклике метод команди, а та – подбає про все інше.



*Доступ з UI до бізнес-логіки через команду.*

Класи команд можна об'єднати під загальним інтерфейсом, що має єдиний метод запуску команди. Після цього одні й ті самі відправники зможуть працювати з різними командами, не прив'язуючись до їхніх класів. Навіть більше, команди можна буде взаємозамінити «на льоту», змінюючи підсумкову поведінку відправників.

Параметри, з якими повинен бути викликаний метод об'єкта отримувача, можна заздалегідь зберегти в полях об'єкта-команди. Завдяки цьому, об'єкти, які надсилають запити, можуть не турбуватися про те, щоб зібрати необхідні дані для отримувача. Навіть більше, вони тепер взагалі не знають, хто буде отримувачем запиту. Вся ця інформація приходить всередині команди.



*Класи UI делегують роботу командам.*

Після застосування Команди в нашому прикладі з текстовим редактором вам більше не потрібно буде створювати безліч підкласів кнопок для різних дій. Буде достатньо одного класу з полем для зберігання об'єкта команди.

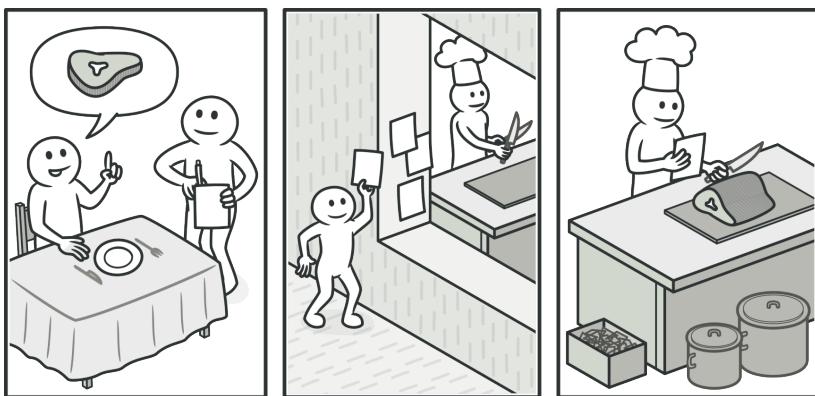
Використовуючи загальний інтерфейс команд, об'єкти кнопок посилатимуться на об'єкти команд різних типів. При натисканні кнопки делегуватимуть роботу командам, а команди – перенаправляти виклики тим чи іншим об'єктам бізнес-логіки.

Так само можна вчинити і з контекстним меню, і з гарячими клавішами. Вони будуть прив'язані до тих самих об'єктів команд, що і кнопки, позбавляючи класи від дублювання.

Таким чином, команди стануть гнучким прошарком між користувачьким інтерфейсом та бізнес-логікою. І це лише

невелика частина тієї користі, яку може принести патерн Команда!

## Аналогія з життя



Приклад замовлення в ресторані.

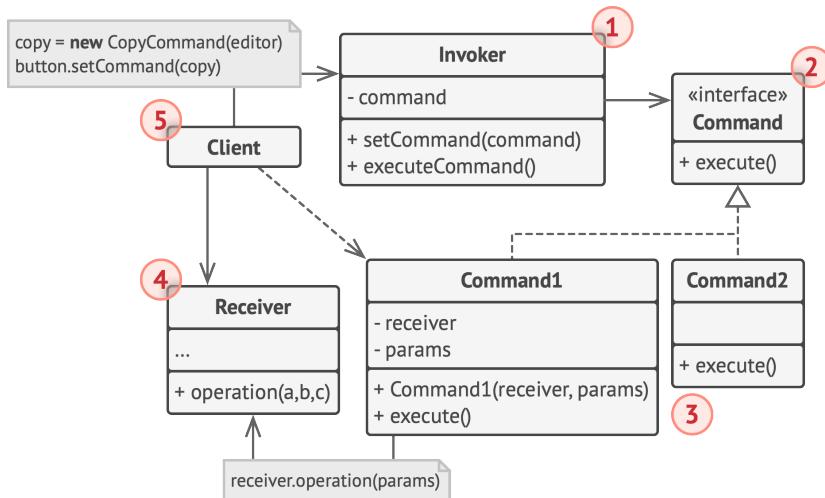
Ви заходите в ресторан і сідаєте біля вікна. До вас підходить ввічливий офіціант і приймає замовлення, записуючи всі побажання в блокнот.

Закінчивши, він поспішає на кухню, вирає аркуш з блокнота та клеїть його на стіну. Далі лист опиняється в руках кухаря, який читає замовлення і готує описану страву.

У цьому прикладі ви є *відправником*, офіціант з блокнотом – *командою*, а кухар – *отримувачем*. Як і в самому патерні, ви не стикаєтесь з кухарем безпосередньо. Замість цього ви відправляєте замовлення офіціантом, який самостійно «налаштовує» кухаря на роботу. З іншого боку, кухар не

знає, хто конкретно надіслав йому замовлення. Але йому це байдуже, бо вся необхідна інформація є в листі замовлення.

## Структура



1. **Відправник** зберігає посилання на об'єкт команди та звертається до нього, коли потрібно виконати якусь дію. Відправник працює з командами тільки через їхній загальний інтерфейс. Він не знає, яку конкретно команду використовує, оскільки отримує готовий об'єкт команди від клієнта.
2. **Команда** описує інтерфейс, спільний для всіх конкретних команд. Зазвичай тут описується лише один метод запуску команди.
3. **Конкретні команди** реалізують різні запити, дотримуючись загального інтерфейсу команд. Як правило, команда не

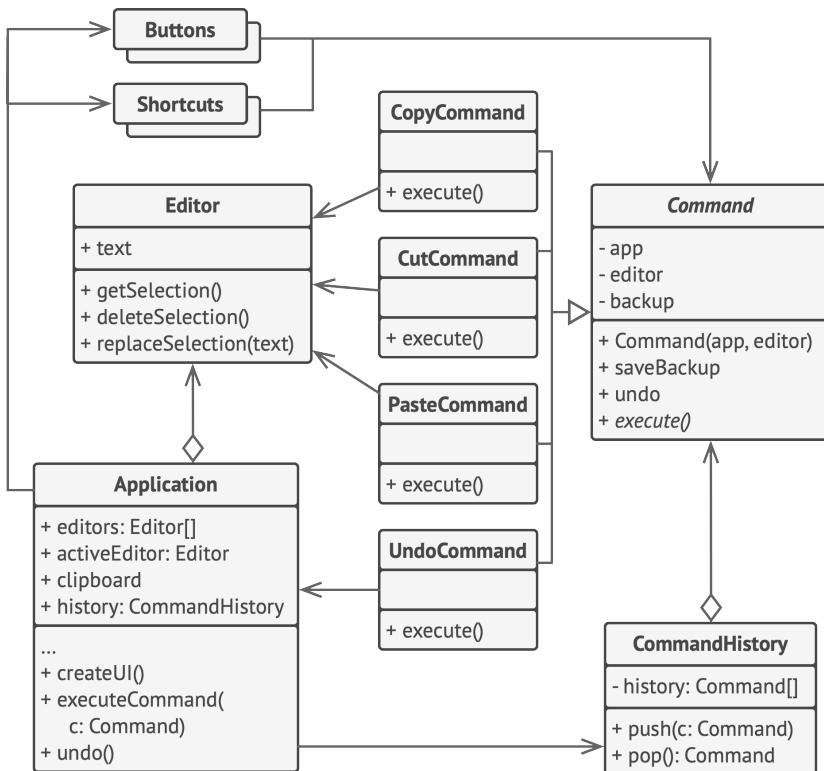
робить всю роботу самостійно, а лише передає виклик одержувачу, яким виступає один з об'єктів бізнес-логіки.

Параметри, з якими команда звертається до одержувача, необхідно зберігати у вигляді полів. У більшості випадків об'єкти команд можна зробити незмінними, передаючи у них всі необхідні параметри тільки через конструктор.

4. **Одержанувач** містить бізнес-логіку програми. У цій ролі може виступати практично будь-який об'єкт. Зазвичай, команди перенаправляють виклики одержувачам, але іноді, щоб спростити програму, ви можете позбутися від одержувачів, «зливши» їхній код у класи команд.
5. **Клієнт** створює об'єкти конкретних команд, передаючи до них усі необхідні параметри, серед яких можуть бути і посилання на об'єкти одержувачів. Після цього клієнт зв'язує об'єкти відправників зі створеними командами.

## # Псевдокод

У цьому прикладі патерн **Команда** використовується для ведення історії виконаних операцій, дозволяючи скасовувати їх за потреби.



*Приклад реалізації скасування у текстовому редакторі.*

Команди, які змінюють стан редактора (наприклад, команда вставки тексту з буфера обміну), зберігають копію стану редактора перед виконанням дії. Копії виконаних команд розміщуються в історії команд, звідки вони можуть бути доставлені, якщо потрібно буде скасувати виконану операцію.

Класи елементів інтерфейсу, історії команд та інші не залежать від конкретних класів команд, оскільки працюють з

ними через загальний інтерфейс. Це дозволяє додавати до програми нові команди, не змінюючи наявний код.

```
1 // Абстрактна команда задає загальний інтерфейс для конкретних
2 // класів команд, а також містить реалізацію базової поведінки
3 // скасування операції.
4 abstract class Command is
5     protected field app: Application
6     protected field editor: Editor
7     protected field backup: text
8
9     constructor Command(app: Application, editor: Editor) is
10         this.app = app
11         this.editor = editor
12
13     // Зберігаємо стан редактора.
14     method saveBackup() is
15         backup = editor.text
16
17     // Відновлюємо стан редактора.
18     method undo() is
19         editor.text = backup
20
21     // Головний метод команди залишається абстрактним, щоб кожна
22     // конкретна команда визначила його по-своєму. Метод повинен
23     // повернути true або false, залежно від того, чи змінила
24     // команда стан редактора, а отже, чи потрібно її зберігати
25     // в історії.
26     abstract method execute()
```

```
29 // Конкретні команди.
30 class CopyCommand extends Command is
31     // Команда копіювання не записується до історії, бо вона не
32     // змінює стан редактора.
33 method execute() is
34     app.clipboard = editor.getSelection()
35     return false
36
37 class CutCommand extends Command is
38     // Команди, що змінюють стан редактора, зберігають стан
39     // редактора перед своєю дією і сигналізують про зміну,
40     // повертуючи true.
41 method execute() is
42     saveBackup()
43     app.clipboard = editor.getSelection()
44     editor.deleteSelection()
45     return true
46
47 class PasteCommand extends Command is
48 method execute() is
49     saveBackup()
50     editor.replaceSelection(app.clipboard)
51     return true
52
53 // Відміна – це також команда.
54 class UndoCommand extends Command is
55 method execute() is
56     app.undo()
57     return false
58
59
60
```

```
61 // Глобальна історія команд – це стек.
62 class CommandHistory is
63     private field history: array of Command
64
65     // Той, що зайшов останнім...
66     method push(c: Command) is
67         // Додати команду в кінець масиву-історії.
68
69     // ...виходить першим.
70     method pop():Command is
71         // Дістати останню команду з масиву-історії.
72
73
74     // Клас редактора містить безпосередні операції над текстом. Він
75     // відіграє роль одержувача – команди делегують йому свої дії.
76 class Editor is
77     field text: string
78
79     method getSelection() is
80         // Повернути вибраний текст.
81
82     method deleteSelection() is
83         // Видалити вибраний текст.
84
85     method replaceSelection(text) is
86         // Вкласти текст з буфера обміну в поточній позиції.
87
88
89     // Клас програми налаштовує об'єкти для спільної роботи. Він
90     // виступає у ролі відправника – створює команди, щоб виконати
91     // якісь дії.
92 class Application is
```

```
93  field clipboard: string
94  field editors: array of Editors
95  field activeEditor: Editor
96  field history: CommandHistory
97
98 // Код, що прив'язує команди до елементів інтерфейсу, може
99 // виглядати приблизно так.
100 method createUI() is
101    // ...
102    copy = function() {executeCommand(
103        new CopyCommand(this, activeEditor)) }
104    copyButton.setCommand(copy)
105    shortcuts.onKeyPress("Ctrl+C", copy)
106
107    cut = function() { executeCommand(
108        new CutCommand(this, activeEditor)) }
109    cutButton.setCommand(cut)
110    shortcuts.onKeyPress("Ctrl+X", cut)
111
112    paste = function() { executeCommand(
113        new PasteCommand(this, activeEditor)) }
114    pasteButton.setCommand(paste)
115    shortcuts.onKeyPress("Ctrl+V", paste)
116
117    undo = function() { executeCommand(
118        new UndoCommand(this, activeEditor)) }
119    undoButton.setCommand(undo)
120    shortcuts.onKeyPress("Ctrl+Z", undo)
121
122 // Запускаємо команду й перевіряємо, чи потрібно додати її
123 // до історії.
124 method executeCommand(command) is
```

```

125     if (command.execute())
126         history.push(command)
127
128     // Беремо останню команду з історії та змушуємо її все
129     // скасувати. Ми не знаємо конкретний тип команди, але це і
130     // не важливо, оскільки кожна команда знає, як скасувати
131     // свою дію.
132     method undo() is
133         command = history.pop()
134         if (command != null)
135             command.undo()

```

## Застосування

 Якщо ви хочете параметризувати об'єкти виконуваною дією.

 Команда перетворює операції на об'єкти, а об'єкти, у свою чергу, можна передавати, зберігати та взаємозамінити всередині інших об'єктів.

Скажімо, ви розробляєте бібліотеки графічного меню і хочете, щоб користувачі могли використовувати меню в різних програмах, не змінюючи кожного разу код ваших класів. Застосувавши патерн, користувачам не доведеться змінювати класи меню, замість цього вони будуть конфігурувати об'єкти меню різними командами.

 Якщо ви хочете поставити операції в чергу, виконувати їх за розкладом або передавати мережею.

 Як і будь-які інші об'єкти, команди можна серіалізувати, тобто перетворити на рядок, щоб потім зберегти у файл або базу даних. Потім в будь-який зручний момент його можна дістати назад, знову перетворити на об'єкт команди та виконати. Так само команди можна передавати мережею, логувати або виконувати на віддаленому сервері.

### Якщо вам потрібна операція скасування.

 Головна річ, яка потрібна для того, щоб мати можливість скасовувати операції – це зберігання історії. Серед багатьох способів реалізації цієї можливості патерн Команда є, мабуть, найпопулярнішим.

Історія команд виглядає як стек, до якого потрапляють усі виконані об'єкти команд. Кожна команда перед виконанням операції зберігає поточний стан об'єкта, з яким вона працюватиме. Після виконання операції копія команди потрапляє до стеку історії, продовжуючи нести у собі збережений стан об'єкта. Якщо знадобиться скасування, програма візьме останню команду з історії та відновить збережений у ній стан.

Цей спосіб має дві особливості. По-перше, точний стан об'єктів не дуже просто зберегти, адже його частина може бути приватною. Вирішити це можна за допомогою патерна Знімок.

По-друге, копії стану можуть займати досить багато операцівної пам'яті. Тому іноді можна вдатися до альтернативної реалізації, тобто замість відновлення старого стану, команда виконає зворотню дію. Недолік цього способу у складності (іноді неможливості) реалізації зворотньої дії.

## Кроки реалізації

1. Створіть загальний інтерфейс команд і визначте в ньому метод запуску.
2. Один за одним створіть класи конкретних команд. У кожному класі має бути поле для зберігання посилання на один або декілька об'єктів-одержувачів, яким команда перенаправлятиме основну роботу.

Крім цього, команда повинна мати поля для зберігання параметрів, потрібних під час виклику методів одержувача. Значення всіх цих полів команда повинна отримувати через конструктор.

І, нарешті, реалізуйте основний метод команди, викликаючи в ньому ті чи інші методи одержувача.

3. Додайте до класів відправників поля для зберігання команд. Зазвичай об'єкти-відправники приймають готові об'єкти команд ззовні – через конструктор або через сетер поля команди.

4. Змініть основний код відправників так, щоб вони делегували виконання дії команді.
  
5. Порядок ініціалізації об'єктів повинен виглядати так:
  - Створюємо об'єкти одержувачів.
  - Створюємо об'єкти команд, зв'язавши їх з одержувачами.
  - Створюємо об'єкти відправників, зв'язавши їх з командами.

## Переваги та недоліки

- ✓ Приирає пряму залежність між об'єктами, що викликають операції, та об'єктами, які їх безпосередньо виконують.
- ✓ Дозволяє реалізувати просте скасування і повтор операцій.
- ✓ Дозволяє реалізувати відкладений запуск операцій.
- ✓ Дозволяє збирати складні команди з простих.
- ✓ Реалізує *принцип відкритості/закритості*.
  
- ✗ Ускладнює код програми внаслідок введення великої кількості додаткових класів.

## Відносини з іншими патернами

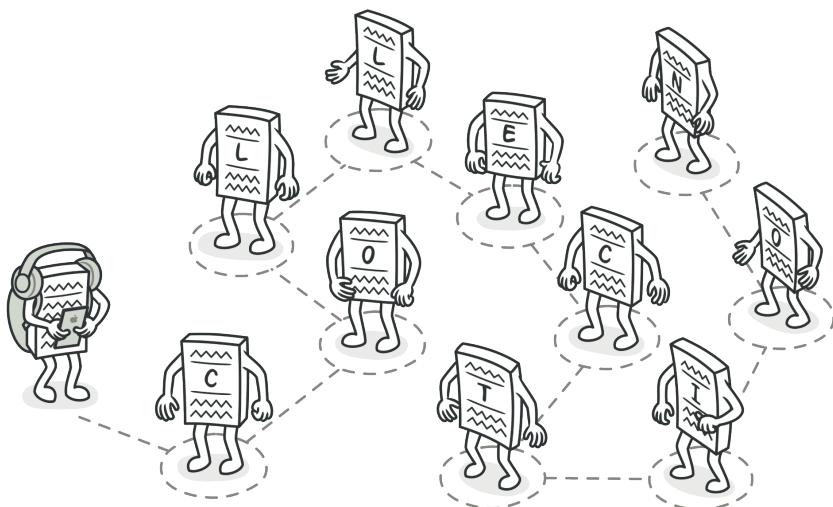
- Ланцюжок обов'язків, Команда Посередник та Спостерігач показують різні способи роботи тих, хто надсилає запити, та тих, хто їх отримує:

- *Ланцюжок обов'язків* передає запит послідовно через ланцюжок потенційних отримувачів, очікуючи, що один з них обробить запит.
- *Команда* встановлює непрямий односторонній зв'язок від відправників до одержувачів.
- *Посередник* прибирає прямий зв'язок між відправниками та одержувачами, змушуючи їх спілкуватися опосередковано, через себе.
- *Спостерігач* передає запит одночасно всім зацікавленим одержувачам, але дозволяє їм динамічно підписуватися або відписуватися від таких повідомлень.
- Обробники в **Ланцюжкові обов'язків** можуть бути виконані у вигляді **Команд**. В цьому випадку роль запиту відіграє контекст команд, який послідовно подається доожної команди у ланцюгу.

Але є й інший підхід, в якому сам запит є *Командою*, надісланою ланцюжком об'єктів. У цьому випадку одна і та сама операція може бути застосована до багатьох різних контекстів, представлених у вигляді ланцюжка.

- **Команду** та **Знімок** можна використовувати спільно для реалізації скасування операцій. У цьому випадку об'єкти команд відповідатимуть за виконання дії над об'єктом, а знімки зберігатимуть резервну копію стану цього об'єкта, зроблену перед запуском команди.

- **Команда** та **Стратегія** схожі за принципом, але відрізняються масштабом та застосуванням:
  - *Команду* використовують для перетворення будь-яких різнопідвидів дій на об'єкти. Параметри операції перетворюються на поля об'єкта. Цей об'єкт тепер можна логувати, зберігати в історії для скасування, передавати у зовнішні сервіси тощо.
  - З іншого боку, *Стратегія* описує різні способи того, як зробити одну і ту саму дію, дозволяючи замінювати ці способи в якомусь об'єкті контексту прямо під час виконання програми.
- Якщо **Команду** потрібно копіювати перед вставкою в історію виконаних команд, вам може допомогти **Прототип**.
- **Відвідувач** можна розглядати як розширений аналог **Команди**, що здатен працювати відразу з декількома видами одержувачів.



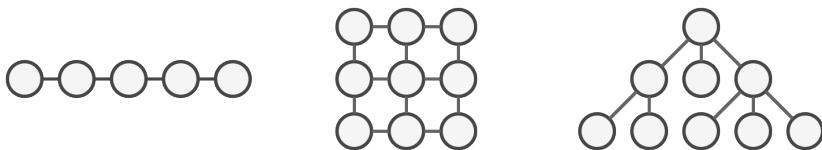
# ІТЕРАТОР

Також відомий як: *Iterator*

**Ітератор** – це поведінковий патерн проектування, що дає змогу послідовно обходити елементи складових об'єктів, не розкриваючи їхньої внутрішньої організації.

## :( Проблема

Колекції – це найпоширеніша структура даних, яку ви можете зустріти в програмуванні. Це набір об'єктів, зібраний в одну купу за якимись критеріями.

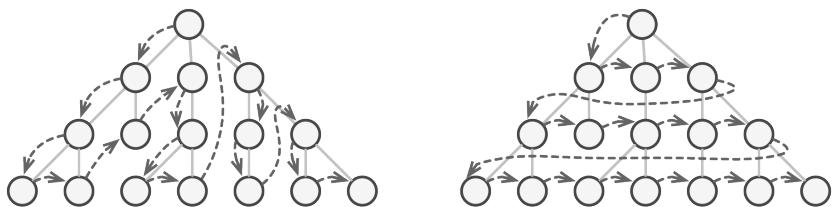


*Різні типи колекцій.*

Більшість колекцій виглядають як звичайний список елементів. Але є й екзотичні колекції, побудовані на основі дерев, графів та інших складних структур даних.

Незважаючи на те, яким чином структуровано колекцію, користувач повинен мати можливість послідовно обходити її елементи, щоб виконувати з ними певні дії.

У який же спосіб слід переміщатися складною структурою даних? Наприклад, сьогодні може бути достатнім обхід дерева в глибину, але завтра виникне необхідність переміщуватися деревом по ширині. А на наступному тижні, хай йому грець, знадобиться можливість обходу колекції у випадковому порядку.



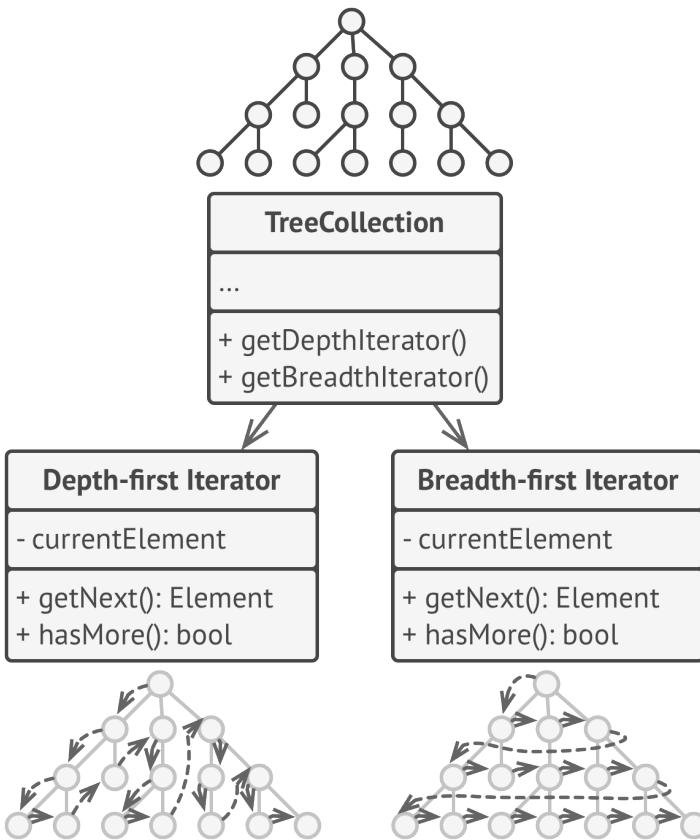
Одну і ту саму колекцію можна обходити різними способами.

Додаючи все нові алгоритми до коду колекції, ви потроху розмиваєте її основну задачу, що полягає в ефективному зберіганні даних. Деякі алгоритми можуть бути аж занадто «заточенні» під певну програму, а тому виглядатимуть неприродно в загальному класі колекції.

## 😊 Рішення

Ідея патерна Ітератор полягає в тому, щоб винести поведінку обходу колекції з самої колекції в окремий об'єкт.

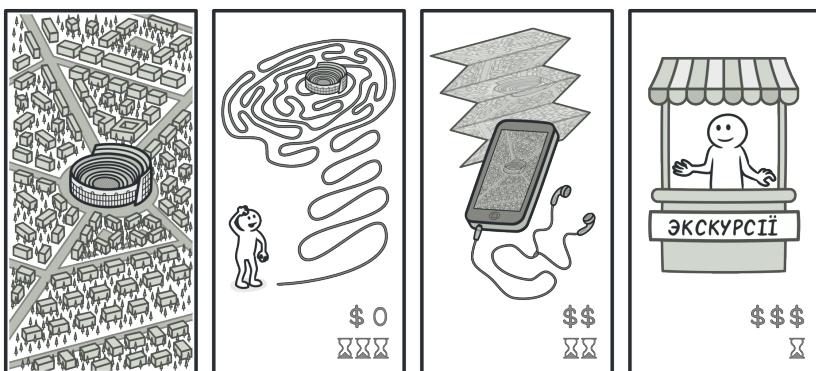
Об'єкт-ітератор відстежуватиме стан обходу, поточну позицію в колекції та кількість елементів, які ще залишилося обійти. Одну і ту саму колекцію зможуть одночасно обходити різні ітератори, а сама колекція навіть не знатиме про це.



*Ітератори містянять код обходу колекції. Одну колекцію можуть обходити відразу декілька ітераторів.*

До того ж, якщо вам потрібно буде додати новий спосіб обходу, ви зможете створите окремий клас ітератора, не змінюючи існуючого коду колекції.

## Аналогія з життя



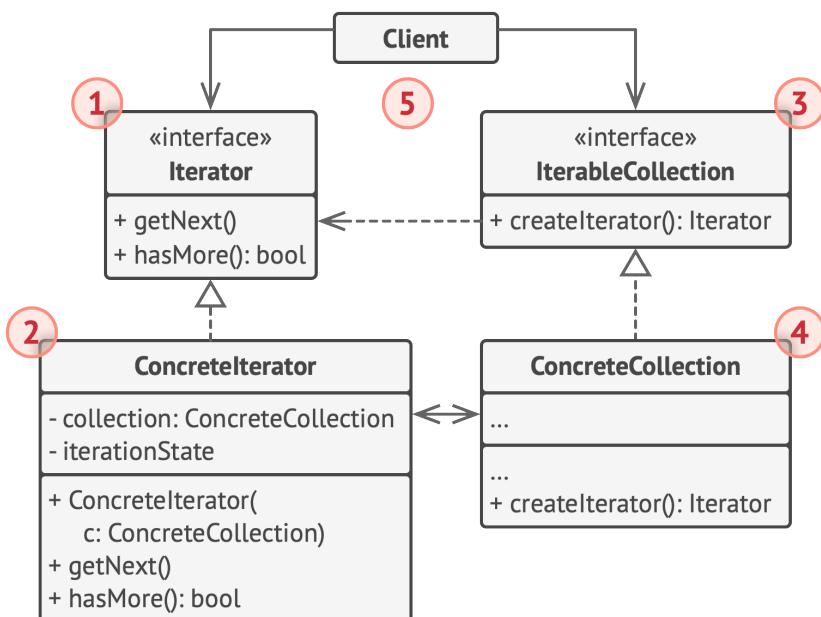
*Варіанти прогулянок Римом.*

Ви плануєте полетіти до Риму та обійти всі визначні пам'ятки за кілька днів. Але по приїзді ви можете довго блукати вузькими вуличками, намагаючись знайти один тільки Колізей.

Якщо у вас обмежений бюджет, ви можете скористатися віртуальним гідом, встановленим у смартфоні, який дозволить відфільтрувати тільки цікаві вам об'єкти. А можете плюнути на все та найняти місцевого гіда, який хоч і обійдеться в копієчку, але знає все місто, як свої п'ять пальців, і зможе «занурити» вас в усі міські легенди.

Таким чином, Рим виступає колекцією пам'яток, а ваш мозок, навігатор чи гід – ітератором колекції. Ви як клієнтський код можете вибрати одного з ітераторів, відштовхуючись від вирішуваного завдання та доступних ресурсів.

## Структура



1. **Ітератор** описує інтерфейс для доступу та обходу елементів колекцій.
2. **Конкретний ітератор** реалізує алгоритм обходу якоїсь конкретної колекції. Об'єкт ітератора повинен сам відстежувати поточну позицію при обході колекції, щоб окремі ітератори могли обходити одну і ту саму колекцію незалежно.
3. **Колекція** описує інтерфейс отримання ітератора з колекції. Як ми вже говорили, колекції не завжди є списком. Це може бути і база даних, і віддалене API, і навіть дерево Компонувальника. Тому сама колекція може створювати ітератори.

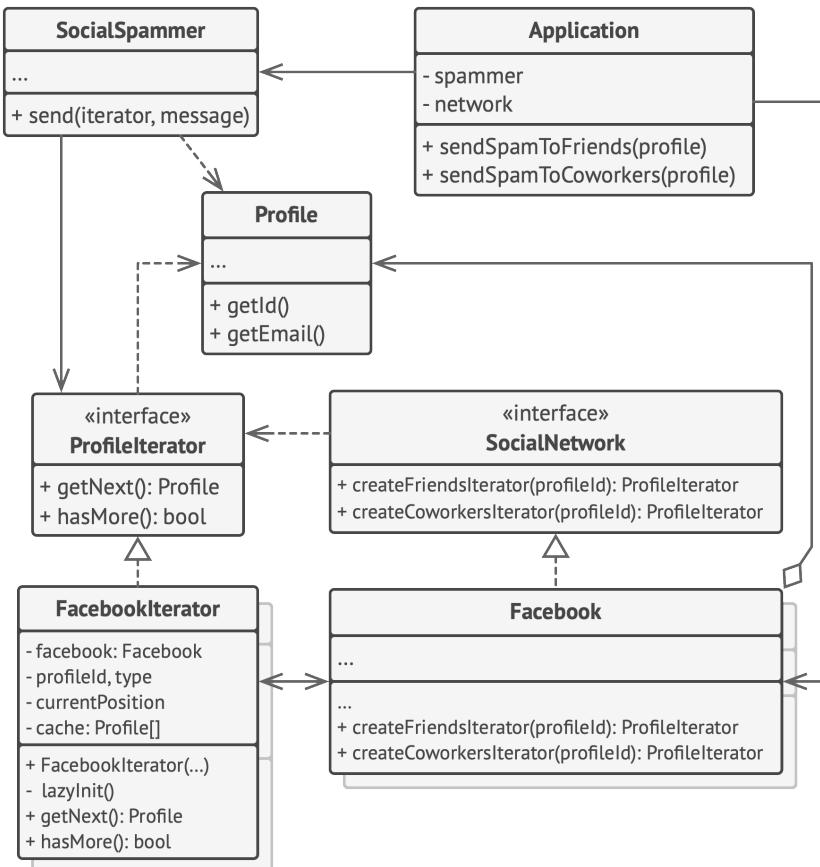
ри, оскільки вона знає, які саме ітератори здатні з нею працювати.

4. **Конкретна колекція** повертає новий екземпляр певного конкретного ітератора, зв'язавши його з поточним об'єктом колекції. Зверніть увагу на те, що сигнатура методу повертає інтерфейс ітератора. Це дозволяє клієнтові не залежати від конкретних класів ітераторів.
5. **Клієнт** працює з усіма об'єктами через інтерфейси колекції та ітератора. Через це клієнтський код не залежить від конкретних класів, що дозволяє застосовувати різні ітератори, не змінюючи існуючого коду програми.

В загальному випадку клієнти не створюють об'єкти ітераторів, а отримують їх з колекцій. Тим не менше, якщо клієнтові потрібний спеціальний ітератор, він завжди може створити його самостійно.

## # Псевдокод

У цьому прикладі патерн **Ітератор** використовується для реалізації обходу нестандартної колекції, яка інкапсулює доступ до соціального графа Facebook. Колекція надає декілька ітераторів, які можуть обходити профілі людей різними способами.



Приклад обходу соціальних профілів через ітератор.

Зокрема, ітератор друзів перебирає всіх друзів профілю, а ітератор колег фільтрує друзів згідно їхньої принадлежності до компанії профілю. Всі ітератори реалізують спільний інтерфейс, який дає змогу клієнтам працювати з профілями, не заглиблюючись у деталі роботи з соціальною мережею (наприклад, авторизацію, надсилання REST запитів та інше).

Крім того, Ітератор позбавляє код від прив'язки до конкретних класів колекцій. Це дозволяє додати підтримку іншого виду колекцій (наприклад, `LinkedIn`), не змінюючи клієнтський код, який працює з ітераторами та колекціями.

```

1 // Загальний інтерфейс колекцій повинен визначити фабричний
2 // метод для виробництва ітератора. Можна визначити відразу
3 // кілька методів, щоб дати користувачам різні варіанти обходу
4 // однієї і тієї самої колекції.
5 interface SocialNetwork is
6     method createFriendsIterator(profileId): ProfileIterator
7     method createCoworkersIterator(profileId): ProfileIterator
8
9
10 // Конкретна колекція знає, об'єкти яких ітераторів потрібно
11 // створювати.
12 class Facebook implements SocialNetwork is
13     // ... Основний код колекції ...
14
15     // Код отримання потрібного ітератора.
16     method createFriendsIterator(profileId) is
17         return new FacebookIterator(this, profileId, "friends")
18     method createCoworkersIterator(profileId) is
19         return new FacebookIterator(this, profileId, "coworkers")
20
21
22 // Загальний інтерфейс ітераторів.
23 interface ProfileIterator is
24     method getNext(): Profile
25     method hasMore(): bool
26

```

```
27 // Конкретний ітератор.
28 class FacebookIterator implements ProfileIterator is
29     // Ітератору потрібне посилання на колекцію, яку він
30     // обходить.
31     private field facebook: Facebook
32     private field profileId, type: string
33
34     // Кожен ітератор обходить колекцію, незалежно від інших,
35     // тому самостійно відслідковує поточну позицію обходу.
36     private field currentPosition
37     private field cache: array of Profile
38
39 constructor FacebookIterator(facebook, profileId, type) is
40     this.facebook = facebook
41     this.profileId = profileId
42     this.type = type
43
44     private method lazyInit() is
45         if (cache == null)
46             cache = facebook.socialGraphRequest(profileId, type)
47
48     // Всі конкретні ітератори реалізують методи загального
49     // інтерфейсу по-своєму.
50     method getNext() is
51         if (hasMore())
52             currentPosition++
53             return cache[currentPosition]
54
55     method hasMore() is
56         lazyInit()
57         return currentPosition < cache.length
58
```

```

59
60 // Ось іще корисна тактика: ми можемо передавати об'єкт
61 // ітератора замість колекції до клієнтських класів. При такому
62 // підході клієнтський код не матиме доступу до колекцій, а
63 // значить, його не турбуватимуть подробиці їхньої реалізації.
64 // Йому буде доступний лише загальний інтерфейс ітераторів.
65 class SocialSpammer is
66     method send(iterator: ProfileIterator, message: string) is
67         while iterator.hasMore()
68             profile = iterator.getNext()
69             System.sendEmail(profile.getEmail(), message)
70
71
72 // Головний клас програми конфігурує ітератори та колекції, як
73 // завгодно.
74 class Application is
75     field network: SocialNetwork
76     field spammer: SocialSpammer
77
78     method config() is
79         if working with Facebook
80             this.network = new Facebook()
81         if working with LinkedIn
82             this.network = new LinkedIn()
83         this.spammer = new SocialSpammer()
84
85     method sendSpamToFriends(profile) is
86         iterator = network.createFriendsIterator(profile.getId())
87         spammer.send(iterator, "Very important message")
88
89     method sendSpamToCoworkers(profile) is
90         iterator = network.createCoworkersIterator(profile.getId())

```

```
91 spammer.send(iterator, "Very important message")
```

## 💡 Застосування

- 💡 Якщо у вас є складна структура даних, і ви хочете приховати від клієнта деталі її реалізації (з питань складності або безпеки).
- ⚡ Ітератор надає клієнтові лише кілька простих методів перебору елементів колекції. Це не тільки спрощує доступ до колекції, але й захищає її від необережних або злонічних дій.
- 💡 Якщо вам потрібно мати кілька варіантів обходу однієї і тієї самої структури даних.
- ⚡ Нетривіальні алгоритми обходу структури даних можуть мати досить об'ємний код. Цей код буде захаращувати все навколо – чи то самий клас колекції, чи частина бізнес-логіки програми. Застосувавши ітератор, ви можете виділити код обходу структури даних в окремий клас, спростивши підтримку решти коду.
- 💡 Якщо вам хочеться мати єдиний інтерфейс обходу різних структур даних.

⚡ Ітератор дозволяє винести реалізації різних варіантів обходу в підкласи. Це дозволить легко взаємозамінити об'єкти ітераторів в залежності від того, з якою структурою даних доводиться працювати.

## Кроки реалізації

1. Створіть загальний інтерфейс ітераторів. Обов'язковий мінімум – це операція отримання наступного елемента. Але для зручності можна передбачити й інше. Наприклад, методи отримання попереднього елементу, поточної позиції, перевірки закінчення обходу тощо.
2. Створіть інтерфейс колекції та опишіть у ньому метод отримання ітератора. Важливо, щоб сигнатура методу повертала загальний інтерфейс ітераторів, а не один з конкретних ітераторів.
3. Створіть класи конкретних ітераторів для тих колекцій, які потрібно обходити за допомогою патерна. Ітератор повинен бути прив'язаний тільки до одного об'єкта колекції. Зазвичай цей зв'язок встановлюється через конструктор.
4. Реалізуйте методи отримання ітератора в конкретних класах колекцій. Вони повинні створювати новий ітератор того класу, який здатен працювати з даним типом колекції. Колекція повинна передавати посилання на власний об'єкт до конструктора ітератора.

- У клієнтському коді та в класах колекцій не повинно залишитися коду обходу елементів. Клієнт повинен отримувати новий ітератор з об'єкта колекції кожного разу, коли йому потрібно перебрати її елементи.

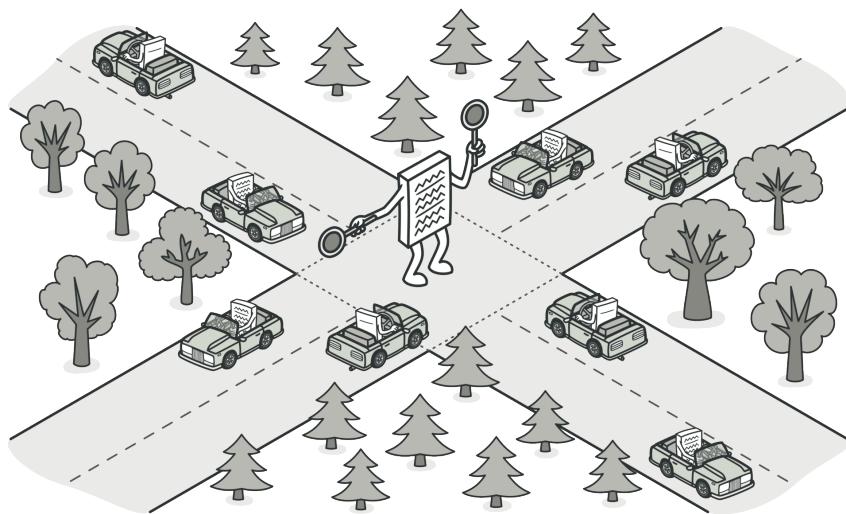
## **Переваги та недоліки**

- ✓ Спрошує класи зберігання даних.
- ✓ Дозволяє реалізувати різні способи обходу структури даних.
- ✓ Дозволяє одночасно переміщуватися структурою даних у різних напрямках.
- ✗ Невиправданий, якщо можна обійтися простим циклом.

## **Відносини з іншими патернами**

- Ви можете обходити дерево Компонувальника, використовуючи Ітератор.
- Фабричний метод можна використовувати разом з Ітератором, щоб підкласи колекцій могли створювати необхідні їм ітератори.
- Знімок можна використовувати разом з Ітератором, щоб зберегти поточний стан обходу структури даних та повернутися до нього в майбутньому, якщо буде потрібно.

- **Відвідувач** можна використовувати спільно з Ітератором.  
*Ітератор* відповідатиме за обхід структури даних, а *Відвідувач* – за виконання дій над кожним її компонентом.



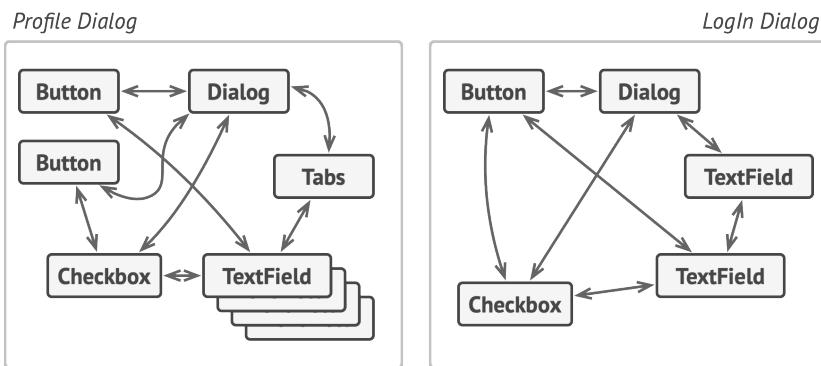
# ПОСЕРЕДНИК

Також відомий як: *Intermediary, Controller, Mediator*

**Посередник** – це поведінковий патерн проектування, що дає змогу зменшити зв'язаність великої кількості класів між собою, завдяки переміщенню цих зв'язків до одного класу-посередника.

## :( Проблема

Припустімо, що у вас є діалог створення профілю користувача. Він складається з різноманітних елементів керування: текстових полів, чекбоксів, кнопок.



*Безладні зв'язки між елементами інтерфейсу користувача.*

Окремі елементи діалогу повинні взаємодіяти одно з одним. Так, наприклад, чекбокс «у мене є собака» відкриває приховане поле для введення імені домашнього улюблена, а клік по кнопці збереження запускає перевірку значень усіх полів форми.

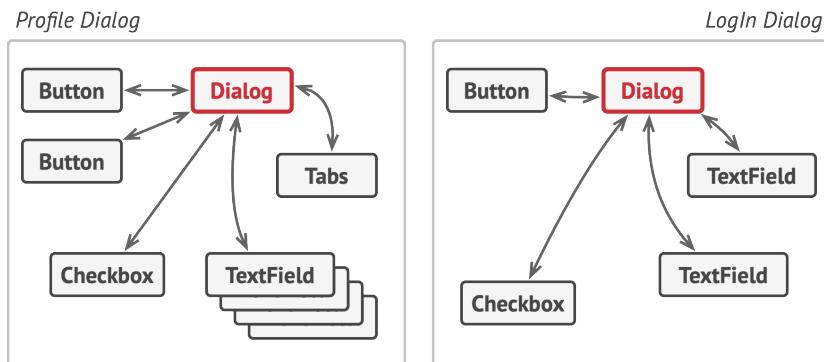


*Код елементів потрібно правити під час зміни кожного діалогу.*

Прописавши цю логіку безпосередньо в коді елементів керування, ви поставите хрест на їхньому повторному використанні в інших місцях програми. Вони стануть занадто тісно пов'язаними з елементами діалогу редагування профілю, які не потрібні в інших контекстах. Отже ви зможете або використовувати всі елементи відразу, або не використовувати жоден.

## 😊 Рішення

Патерн Посередник змушує об'єкти спілкуватися через окремий об'єкт-посередник, який знає, кому потрібно перенаправити той або інший запит. Завдяки цьому компоненти системи залежатимуть тільки від посередника, а не від десятків інших компонентів.



*Елементи інтерфейсу спілкуються через посередника.*

У нашому прикладі посередником міг би стати діалог. Імовірно, клас діалогу вже знає, з яких елементів він скла-

дається. Тому жодних нових зв'язків додавати до нього не доведеться.

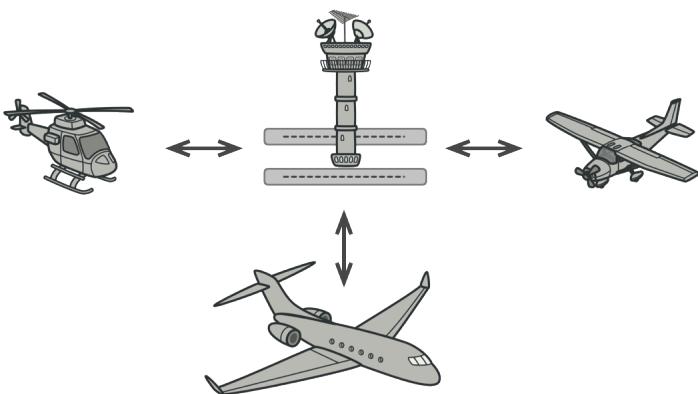
Основні зміни відбудуться всередині окремих елементів діалогу. Якщо раніше при отриманні кліка від користувача об'єкт кнопки самостійно перевіряв значення полів діалогу, то тепер його єдиний обов'язок – повідомити діалогу про те, що відбувся клік. Отримавши повідомлення, діалог виконає всі необхідні перевірки полів. Таким чином, замість кількох залежностей від інших елементів кнопка отримає лише одну – від самого діалогу.

Щоб зробити код ще гнучкішим, можна виділити єдиний інтерфейс для всіх посередників, тобто діалогів програми. Наша кнопка стане залежною не від конкретного діалогу створення користувача, а від абстрактного, що дозволить використовувати її і в інших діалогах.

Таким чином, посередник приховує у собі всі складні зв'язки й залежності між класами окремих компонентів програми. А чим менше зв'язків мають класи, тим простіше їх змінювати, розширювати й повторно використовувати.

## Аналогія з життя

Пілоти літаків, що сідають або злітають, не спілкуються з іншими пілотами безпосередньо. Замість цього вони зв'язуються з диспетчером, який координує політ кількох літаків одночасно.

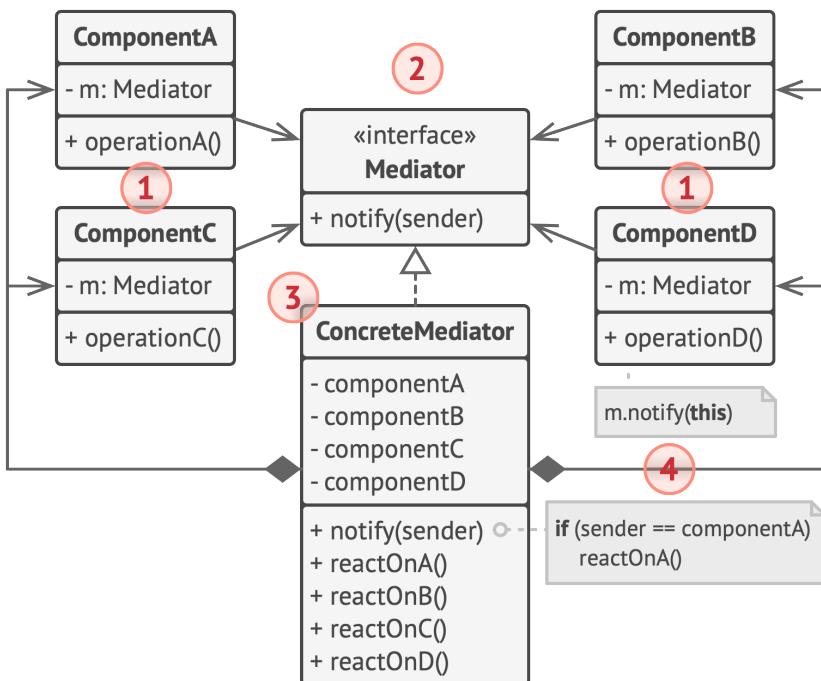


*Пілоти літаків спілкуються не безпосередньо, а через диспетчера.*

Без диспетчера пілотам доводилося б увесь час бути напоготові і стежити самостійно за всіма літаками навколо. Це часто призводило б до катастроф у небі.

Важливо розуміти, що диспетчер не потрібен під час всього польоту. Він задіяний тільки в зоні аеропорту, коли потрібно координувати взаємодію багатьох літаків.

## Структура



- Компоненти** – це різнорідні об'єкти, що містять бізнес-логіку програми. Кожен компонент має посилання на об'єкт посередника, але працює з ним тільки через абстрактний інтерфейс посередників. Завдяки цьому компоненти можна повторно використовувати в інших програмах, зв'язавши їх з посередником іншого типу.
- Посередник** визначає інтерфейс для обміну інформацією з компонентами. Зазвичай достатньо одного методу, щоби повідомляти посередника про події, що відбулися в компонентах. У параметрах цього методу можна передавати дета-

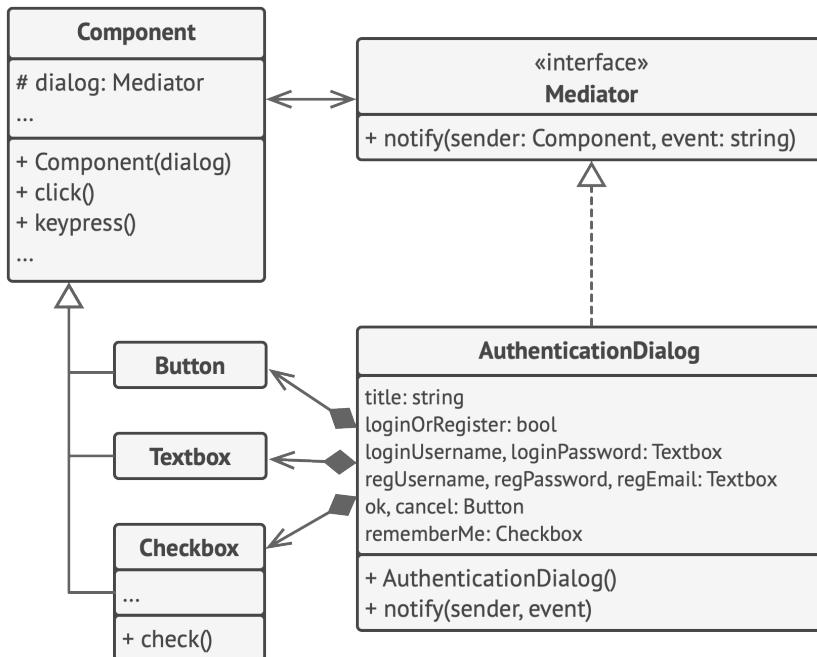
лі події: посилання на компонент, в якому вона відбулася, та будь-які інші дані.

3. **Конкретний посередник** містить код взаємодії кількох компонентів між собою. Найчастіше цей об'єкт не тільки зберігає посилання на всі свої компоненти, але й сам їх створює, керуючи подальшим життєвим циклом.
4. Компоненти не повинні спілкуватися один з одним безпосередньо. Якщо в компоненті відбувається важлива подія, він повинен повідомити свого посередника, а той сам вирішить, чи стосується подія інших компонентів, і чи треба їх сповістити. При цьому компонент-відправник не знає, хто обробить його запит, а компонент-одержувач не знає, хто його надіслав.

## # Псевдокод

У цьому прикладі **Посередник** допомагає позбутися залежностей між класами різних елементів користувача інтерфейсу: кнопками, чекбоксами й написами.

Реагуючи на дії користувачів, елементи не взаємодіють безпосередньо, а лише повідомляють посередника про те, що вони змінилися.



*Приклад структурування класів UI діалогів.*

Посередник у вигляді діалогу авторизації знає, як конкретні елементи повинні взаємодіяти. Тому при отриманні повідомлень він може перенаправити виклик тому чи іншому елементу.

```

1 // Загальний інтерфейс посередників.
2 interface Mediator is
3     method notify(sender: Component, event: string)
4
5
6 // Конкретний посередник. Усі зв'язки між конкретними
7 // компонентами переїхали до коду посередника. Він отримує

```

```
8 // повідомлення від своїх компонентів та знає, як на них
9 // реагувати.
10 class AuthenticationDialog implements Mediator is
11     private field title: string
12     private field loginOrRegisterChkBx: Checkbox
13     private field loginUsername, loginPassword: Textbox
14     private field registrationUsername, registrationPassword,
15             registrationEmail: Textbox
16     private field okBtn, cancelBtn: Button
17
18 constructor AuthenticationDialog() is
19     // Тут потрібно буде створити об'єкти усіх компонентів,
20     // подавши поточний об'єкт-посередник до їхніх
21     // конструкторів.
22
23     // Коли щось трапляється з компонентом, він надсилає
24     // посереднику повідомлення. Після отримання повідомлення
25     // посередник може або зробити щось самостійно, або
26     // перенаправити запит іншому компонентові.
27 method notify(sender, event) is
28     if (sender == loginOrRegisterChkBx and event == "check")
29         if (loginOrRegisterChkBx.checked)
30             title = "Log in"
31             // 1. Показати компоненти форми входу.
32             // 2. Приховати компоненти форми реєстрації.
33     else
34         title = "Register"
35         // 1. Показати компоненти форми реєстрації.
36         // 2. Приховати компоненти форми входу.
37
38     if (sender == okBtn && event == "click")
39         if (loginOrRegister.checked)
```

```
40         // Намагатись знайти користувача з даними із
41         // форми логіна.
42         if (!found)
43             // Показати помилку над формою логіна.
44     else
45         // 1. Створити аккаунт користувача з даними
46         // форми реєстрації.
47         // 2. Авторизувати цього користувача.
48         // ...
49
50
51 // Класи компонентів спілкуються з посередниками через їх
52 // загальний інтерфейс. Завдяки цьому, одні й ті ж компоненти
53 // можна використовувати в різних посередниках.
54 class Component is
55     field dialog: Mediator
56
57     constructor Component(dialog) is
58         this.dialog = dialog
59
60     method click() is
61         dialog.notify(this, "click")
62
63     method keypress() is
64         dialog.notify(this, "keypress")
65
66 // Конкретні компоненти жодним чином не пов'язані між собою. У
67 // них є тільки один канал спілкування – через надсилання
68 // повідомень посереднику.
69 class Button extends Component is
70     // ...
71
```

```
72 class Textbox extends Component is
73     // ...
74
75 class Checkbox extends Component is
76     method check() is
77         dialog.notify(this, "check")
78     // ...
```

## Придатність

- Коли вам складно змінювати деякі класи через те, що вони мають величезну кількість хаотичних зв'язків з іншими класами.
- Посередник дозволяє розмістити усі ці зв'язки в одному класі. Після цього вам буде легше їх відрефакторити, зробити більш зрозумілими й гнучкими.
- Коли ви не можете повторно використовувати клас, оскільки він залежить від безлічі інших класів.
- Після застосування патерна компоненти втрачають колишні зв'язки з іншими компонентами, а все їхнє спілкування відбувається опосередковано, через об'єкт посередника.
- Коли вам доводиться створювати багато підкласів компонентів, щоб використовувати одні й ті самі компоненти в різних контекстах.

- Якщо раніше зміна відносин в одному компоненті могла призвести до лавини змін в усіх інших компонентах, то тепер вам достатньо створити підклас посередника та змінити в ньому зв'язки між компонентами.

## Кроки реалізації

1. Знайдіть групу тісно сплетених класів, де можна отримати деяку користь, відв'язавши деякі один від одного. Наприклад, щоб повторно використовувати їхній код в іншій програмі.
2. Створіть загальний інтерфейс посередників та опишіть в ньому методи для взаємодії з компонентами. У найпростішому випадку достатньо одного методу для отримання повідомлень від компонентів.

Цей інтерфейс необхідний, якщо ви хочете повторно використовувати класи компонентів для інших завдань. У цьому випадку все, що потрібно зробити, – це створити новий клас конкретного посередника.

3. Реалізуйте цей інтерфейс у класі конкретного посередника. Помістіть до нього поля, які міститимуть посилання на всі об'єкти компонентів.
4. Ви можете піти далі і перемістити код створення компонентів до класу конкретного посередника, перетворивши його на фабрику.

5. Компоненти теж повинні мати посилання на об'єкт посередника. Зв'язок між ними зручніше всього встановити шляхом подання посередника до параметрів конструктора компонентів.
6. Змініть код компонентів так, щоб вони викликали метод повідомлення посередника, замість методів інших компонентів. З протилежного боку, посередник має викликати методи потрібного компонента, коли отримує повідомлення від компонента.

## **Переваги та недоліки**

- ✓ Усуває залежності між компонентами, дозволяючи використовувати їх повторно.
- ✓ Спрощує взаємодію між компонентами.
- ✓ Централізує керування в одному місці.
- ✗ Посередник може сильно «роздутися».

## **Відносини з іншими патернами**

- Ланцюжок обов'язків, Команда Посередник та Спостерігач показують різні способи роботи тих, хто надсилає запити, та тих, хто їх отримує:
  - *Ланцюжок обов'язків* передає запит послідовно через ланцюжок потенційних отримувачів, очікуючи, що один з них обробить запит.

- Команда встановлює непрямий односторонній зв’язок від відправників до одержувачів.
- Посередник прибирає прямий зв’язок між відправниками та одержувачами, змушуючи їх спілкуватися опосередковано, через себе.
- Спостерігач передає запит одночасно всім зацікавленим одержувачам, але дозволяє їм динамічно підписуватися або відписуватися від таких повідомлень.
- Посередник та Фасад схожі тим, що намагаються організувати роботу багатьох існуючих класів.
  - Фасад створює спрощений інтерфейс підсистеми, не вносячи в неї жодної додаткової функціональності. Сама підсистема не знає про існування Фасаду. Класи підсистеми спілкуються один з одним безпосередньо.
  - Посередник централізує спілкування між компонентами системи. Компоненти системи знають тільки про існування Посередника, у них немає прямого доступу до інших компонентів.
- Різниця між Посередником та Спостерігачем не завжди очевидна. Найчастіше вони виступають як конкуренти, але іноді можуть працювати разом.

Мета Посередника – прибрати взаємні залежності між компонентами системи. Замість цього вони стають залежними від самого посередника. З іншого боку, мета Спосте-

*рігача* – забезпечити динамічний односторонній зв’язок, в якому одні об’єкти опосередковано залежать від інших.

Досить популярною є реалізація *Посередника* за допомогою *Спостерігача*. При цьому об’єкт посередника буде виступати видавцем, а всі інші компоненти стануть передплатниками та зможуть динамічно стежити за подіями, що відбуваються у посереднику. У цьому випадку важко зрозуміти, чим саме відрізняються обидва патерни.

Але *Посередник* має й інші реалізації, коли окремі компоненти жорстко прив’язані до об’єкта посередника. Такий код навряд чи буде нагадувати *Спостерігача*, але залишиться *Посередником*.

Навпаки, у разі реалізації посередника з допомогою *Спостерігача*, представимо чи уявімо таку програму, в якій кожен компонент системи стає видавцем. Компоненти можуть підписуватися один на одного, не прив’язуючись до конкретних класів. Програма складатиметься з цілої мережі *Спостерігачів*, не маючи центрального об’єкта *Посередника*.



# ЗНІМОК

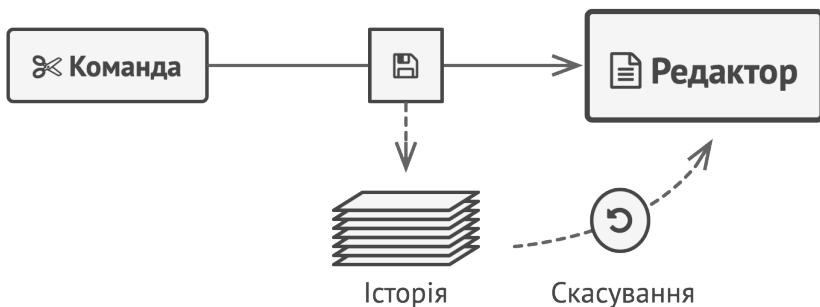
Також відомий як: *Memento*

**Знімок** – це поведінковий патерн проектування, що дає змогу зберігати та відновлювати минулий стан об'єктів, не розкриваючи подробиць їхньої реалізації.

## :( Проблема

Припустімо, ви пишете програму текстового редактора. Крім звичайного редагування, ваш редактор дозволяє змінювати форматування тексту, вставляти малюнки та інше.

В певний момент ви вирішили надати можливість скасовувати усі ці дії. Для цього вам потрібно зберігати поточний стан редактора перед тим, як виконати будь-яку дію. Якщо користувач вирішить скасувати свою дію, ви візьмете копію стану з історії та відновите попередній стан редактора.

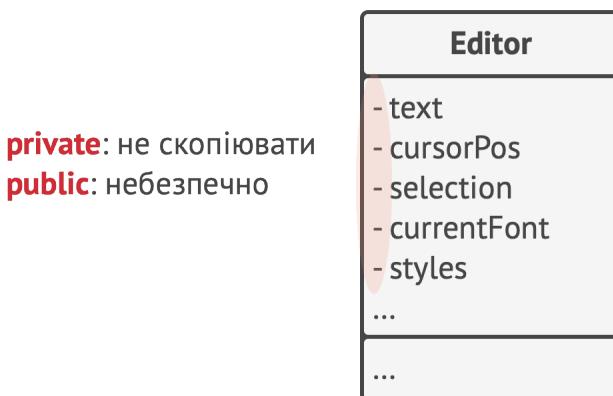


*Перед виконанням команди ви можете зберегти копію стану редактора, щоб потім мати можливість скасувати операцію.*

Щоб зробити копію стану об'єкта, достатньо скопіювати значення полів. Таким чином, якщо ви зробили клас редактора достатньо відкритим, то будь-який інший клас зможе зазирнути всередину, щоб скопіювати його стан.

Здавалося б, які проблеми? Тепер будь-яка операція зможе зробити резервну копію редактора перед виконанням своєї

дії. Але такий наївний підхід забезпечить вам безліч проблем у майбутньому. Адже, якщо ви вирішите провести рефакторинг – прибрати або додати кілька полів до класу редактора – доведеться змінювати код усіх класів, які могли копіювати стан редактора.



*Як команді створити знімок стану редактора, якщо всі його поля приватні?*

Але це ще не все. Давайте тепер поглянемо безпосередньо на копії стану, які ми створювали. З чого складається стан редактора? Навіть найпримітивніший редактор повинен мати декілька полів для зберігання поточного тексту, позиції курсора та прокручування екрану. Щоб зробити копію стану, вам потрібно додати значення всіх цих полів до деякого «контейнера». Імовірно, вам знадобиться зберігати масу таких контейнерів в якості історії операцій, тому зручніше за все зробити їх об'єктами одного класу. Цей клас повинен мати багато полів, але практично жодного методу. Щоб інші об'єкти могли записувати та читати з нього дані, вам дове-

деться зробити його поля публічними. Проте це призведе до тієї ж проблеми, що й з відкритим класом редактора. Інші класи стануть залежними від будь-яких змін класу контейнера, який схильний до таких самих змін, що і клас редактора.

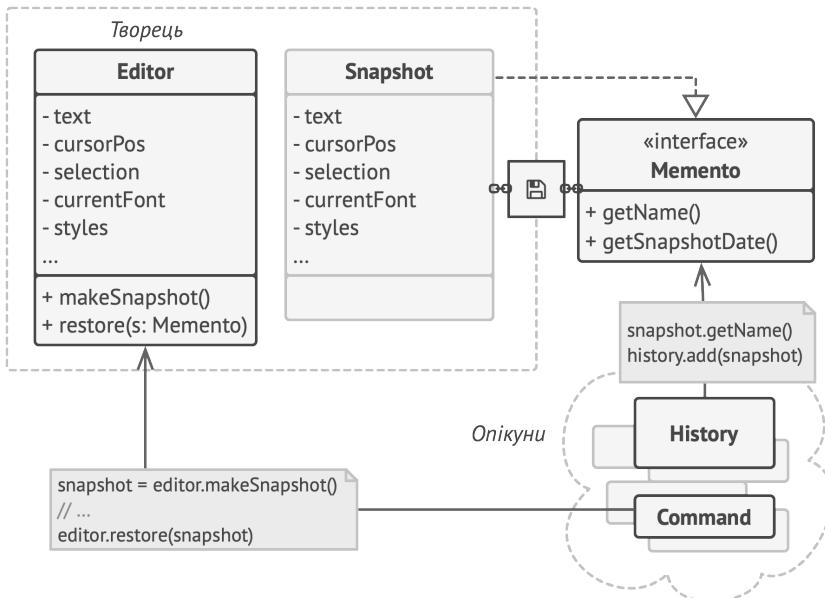
Виходить, що нам доведеться або відкрити класи для всіх бажаючих, отримавши постійний клопіт з підтримкою коду, або залишити класи закритими, відмовившись від ідеї скавування операцій. Чи немає тут альтернативи?

## Рішення

Усі проблеми, описані вище, виникають через порушення інкапсуляції, коли одні об'єкти намагаються зробити роботу за інших, проникаючи до їхньої приватної зони, щоб зібрати необхідні для операції дані.

Патерн Знімок доручає створення копії стану об'єкта самому об'єкту, який цим станом володіє. Замість того, щоб робити знімок «ззовні», наш редактор сам зробить копію своїх полів, адже йому доступні всі поля, навіть приватні.

Патерн пропонує тримати копію стану в спеціальному об'єкті-знімку з обмеженим інтерфейсом, що дозволяє, наприклад, дізнатися дату виготовлення або назву знімка. Проте, знімок повинен бути відкритим для свого *творця* і дозволяти прочитати та відновити його внутрішній стан.



Знімок повністю відкритий для творця, але лише частково відкритий для опікунів.

Така схема дозволяє творцям робити знімки та віддавати їх на зберігання іншим об'єктам, що називаються *опікунами*. Опікунам буде доступний тільки обмежений інтерфейс знімка, тому вони ніяк не зможуть вплинути на «нутрощі» самого знімку. У потрібний момент опікун може попросити творця відновити свій стан, передавши йому відповідний знімок.

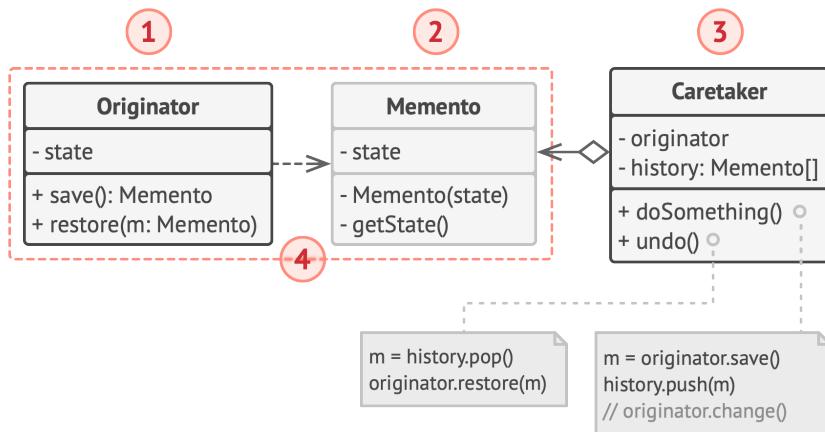
У нашому прикладі з редактором опікуном можна зробити окремий клас, який зберігатиме список виконаних операцій. Обмежений інтерфейс знімків дозволить демонструвати користувачеві гарний список з назвами й датами виконаних операцій. Коли ж користувач вирішить скасувати операцію,

клас історії візьме останній знімок зі стека та надішле його об'єкту редактора для відновлення.

## Структура

### Класична реалізація на вкладених класах

Класична реалізація патерна покладається на механізм вкладених класів, який доступний тільки в деяких мовах програмування (C++, C#, Java).

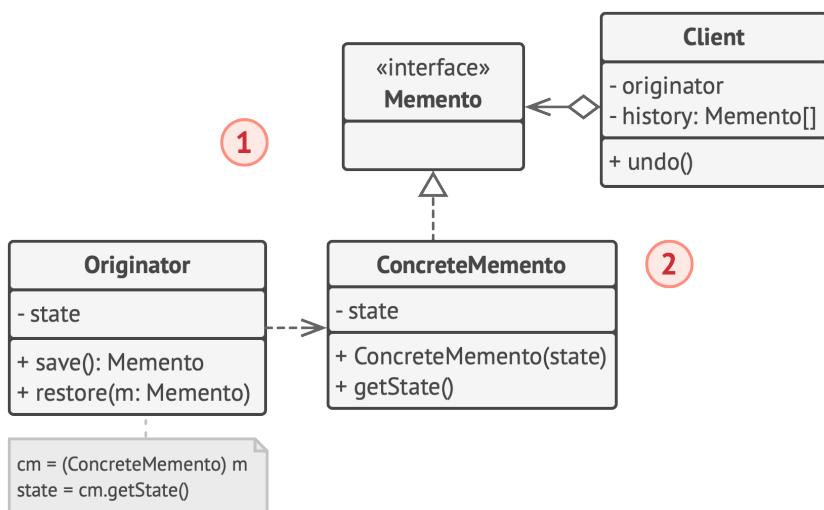


- Творець** може створювати знімки свого стану, а також відтворювати минулий стан, якщо до нього подати готовий знімок.
- Знімок** – це простий об'єкт даних, який містить стан творця. Надійніше за все зробити об'єкти знімків незмінними, встановлюючи в них стан тільки через конструктор.

3. **Опікун** повинен знати, коли робити знімок творця та коли його потрібно відновлювати. Опікун може зберігати історію минулих станів творця у вигляді стека знімків. Коли треба буде скасувати останню операцію, він візьме «верхній» знімок зі стеку та передасть його творцеві для відновлення.
  
4. У даній реалізації знімок – це внутрішній клас по відношенню до класу творця. Саме тому він має повний доступ до всіх полів та методів творця, навіть приватних. З іншого боку, опікун не має доступу ані до стану, ані до методів знімків, а може лише зберігати посилання на ці об'єкти.

### Реалізація з проміжним порожнім інтерфейсом

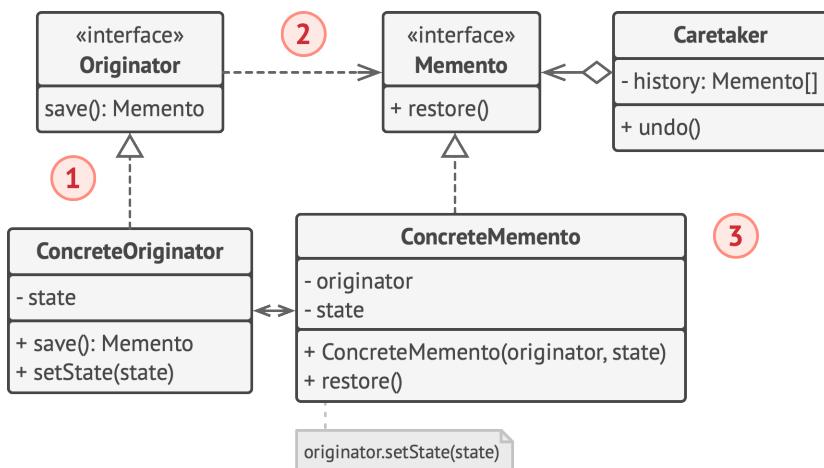
Підходить для мов, що не мають механізму вкладених класів (наприклад, PHP).



- У цій реалізації творець працює безпосередньо з конкретним класом знімка, а опікун – тільки з його обмеженим інтерфейсом.
- Завдяки цьому досягається той самий ефект, що і в класичній реалізації. Творець має повний доступ до знімка, а опікун – ні.

### Знімки з підвищеним захистом

Якщо потрібно повністю виключити можливість доступу до стану творців та знімків.

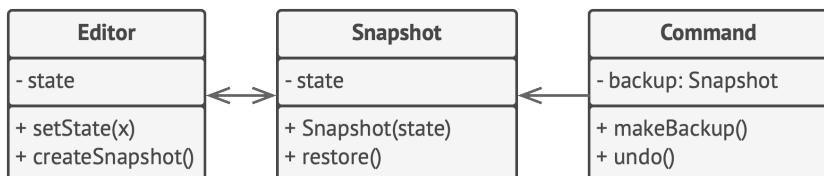


- Ця реалізація дозволяє мати кілька видів творців та знімків. Кожному класу творців відповідає власний клас знімків. Ані творці, ані знімки не дозволяють іншим об'єктам читати свій стан.

2. Тут опікун ще жорсткіше обмежений у доступі до стану творців та знімків, але, з іншого боку, опікун стає незалежним від творців, оскільки метод відновлення тепер знаходиться в самих знімках.
3. Знімки тепер пов'язані з тими творцями, з яких вони зроблені. Вони, як і раніше, отримують стан через конструктор. Завдяки близькому зв'язку між класами, знімки знають, як відновити стан своїх творців.

## # Псевдокод

У цьому прикладі патерн **Знімок** використовується спільно з патерном **Команда** та дозволяє зберігати резервні копії складного стану текстового редактора й відновлювати його за потреби.



*Приклад збереження знімків стану текстового редактора.*

Об'єкти команд виступають в ролі опікунів і запитують знімки в редактора перед тим, як виконати свою дію. Якщо знадобиться скасувати операцію, команда зможе відновити стан редактора, використовуючи збережений знімок.

При цьому знімок не має публічних полів, тому інші об'єкти не мають доступу до його внутрішніх даних. Знімки пов'язані з певним редактором, який їх створив. Вони ж і відновлюють стан свого редактора. Це дозволяє програмі мати одночасно кілька об'єктів редакторів, наприклад, розбитих по різних вкладках програми.

```

1 // Клас творця повинен мати спеціальний метод, який зберігає
2 // стан об'єкта в новому об'єкті-знімку.
3 class Editor is
4     private field text, curX, curY, selectionWidth
5
6     method setText(text) is
7         this.text = text
8
9     method setCursor(x, y) is
10        this.curX = x
11        this.curY = y
12
13    method setSelectionWidth(width) is
14        this.selectionWidth = width
15
16    method createSnapshot(): Snapshot is
17        // Знімок – це незмінний об'єкт, тому творець передає до
18        // нього свій стан через параметри конструктора.
19        return new Snapshot(this, text, curX, curY, selectionWidth)
20
21 // Знімок зберігає минулий стан редактора.
22 class Snapshot is
23     private field editor: Editor
24     private field text, curX, curY, selectionWidth

```

```
25 constructor Snapshot(editor, text, curX, curY, selectionWidth) is
26     this.editor = editor
27     this.text = text
28     this.curX = x
29     this.curY = y
30     this.selectionWidth = selectionWidth
31
32 // У потрібний момент власник знімку може відновити стан
33 // редактора.
34 method restore() is
35     editor.setText(text)
36     editor.setCursor(curX, curY)
37     editor.setSelectionWidth(selectionWidth)
38
39 // Опікуном може виступати клас команд (див. патерн Команда). У
40 // цьому випадку команда зберігає знімок стану об'єкта-
41 // одержувача перед тим, як передати йому дію. А в разі
42 // скасування дії, команда поверне об'єкт до попереднього стану.
43 class Command is
44     private field backup: Snapshot
45
46     method makeBackup() is
47         backup = editor.createSnapshot()
48
49     method undo() is
50         if (backup != null)
51             backup.restore()
52     // ...
```

## Застосування

-  Коли вам потрібно зберігати миттєві знімки стану об'єкта (або його частини) для того, щоб об'єкт можна було відновити в тому самому стані.
-  Патерн Знімок дозволяє створювати будь-яку кількість знімків об'єкта і зберігати їх незалежно від об'єкта, з якого роблять знімок. Знімки часто використовують не тільки для реалізації операції скасування, але й для транзакцій, коли стан об'єкта потрібно «відкотити», якщо операція не була вдалою.
-  Коли пряме отримання стану об'єкта розкриває приватні деталі його реалізації, порушуючи інкапсуляцію.
-  Патерн пропонує виготовити знімок саме вихідному об'єкту, тому що йому доступні всі поля, навіть приватні.

## Кроки реалізації

1. Визначте клас творця, об'єкти якого повинні створювати знімки свого стану.
2. Створіть клас знімка та описіть в ньому ті ж самі поля, які є в оригінальному класі-творці.

3. Зробіть об'єкти знімків незмінними. Вони повинні одержувати початкові значення тільки один раз, через власний конструктор.
4. Якщо ваша мова програмування це дозволяє, зробіть клас знімка вкладеним у клас творця.

Якщо ні, вийміть з класу знімка порожній інтерфейс, який буде доступним іншим об'єктам програми. Згодом ви можете додати до цього інтерфейсу деякі допоміжні методи, що дають доступ до метаданих знімка, але прямий доступ до даних творця повинен бути виключеним.

5. Додайте до класу творця метод одержання знімків. Творець повинен створювати нові об'єкти знімків, передаючи значення своїх полів через конструктор.

Сигнатура методу повинна повернати знімки через обмежений інтерфейс, якщо він у вас є. Сам клас повинен працювати з конкретним класом знімка.

6. Додайте до класу творця метод відновлення зі знімка. Щодо прив'язки до типів, керуйтесь тією ж логікою, що і в пункті 4.
7. Опікуни, незалежно від того, чи це історія операцій, чи об'єкти команд, чи щось інше, повинні знати про те, коли запитувати знімки у творця, де їх зберігати та коли відновлювати.

8. Зв'язок опікунів з творцями можна перенести всередину знімків. У цьому випадку кожен знімок буде прив'язаний до свого творця і повинен буде сам відновлювати його стан. Але це працюватиме або якщо класи знімків вкладені до класів творців, або якщо творці мають відповідні сетери для встановлення значень своїх полів.

## **Переваги та недоліки**

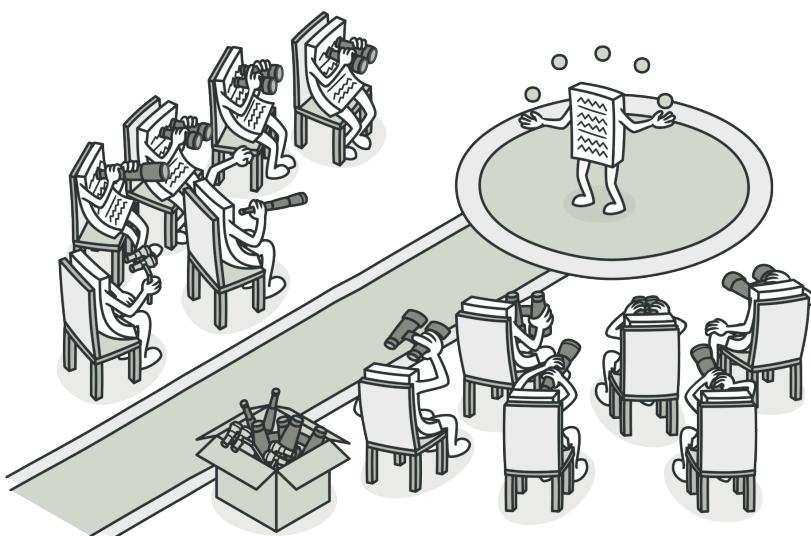
- ✓ Не порушує інкапсуляцію вихідного об'єкта.
- ✓ Спрощує структуру вихідного об'єкта. Йому не потрібно зберігати історію версій свого стану.
- ✗ Вимагає багато пам'яті, якщо клієнти дуже часто створюють знімки.
- ✗ Може спричинити додаткові витрати пам'яті, якщо об'єкти, що зберігають історію, не звільняють ресурси, зайняті застарілими знімками.
- ✗ В деяких мовах (наприклад, PHP, Python, JavaScript) складно гарантувати, щоб лише вихідний об'єкт мав доступ до стану знімка.

## **Відносини з іншими патернами**

- **Команду** та **Знімок** можна використовувати спільно для реалізації скасування операцій. У цьому випадку об'єкти команда відповідатимуть за виконання дії над об'єктом, а знімки

зберігатимуть резервну копію стану цього об'єкта, зроблену перед запуском команди.

- Знімок можна використовувати разом з Ітератором, щоб зберегти поточний стан обходу структури даних та повернутися до нього в майбутньому, якщо буде потрібно.
- Знімок іноді можна замінити Прототипом, якщо об'єкт, чий стан потрібно зберігати в історії, досить простий, не має посилань на зовнішні ресурси або їх можна легко відновити.



# СПОСТЕРІГАЧ

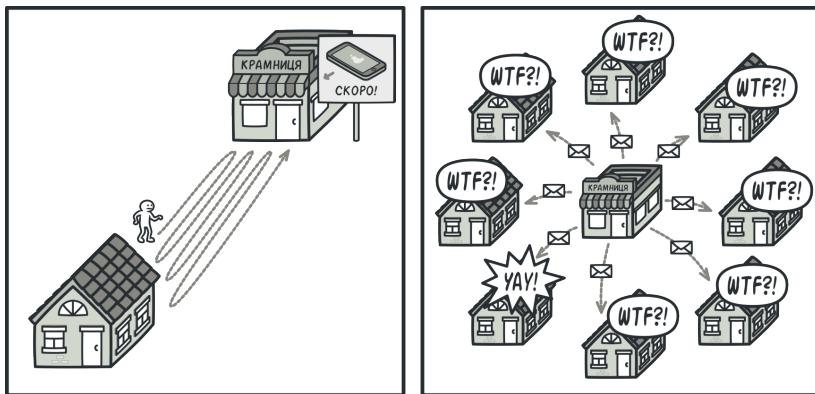
Також відомий як: Видавець-Підписник, Слухач, *Observer*

**Спостерігач** – це поведінковий патерн проектування, який створює механізм підписки, що дає змогу одним об'єктам стежити й реагувати на події, які відбуваються в інших об'єктах.

## (:() Проблема

Уявіть, що ви маєте два об'єкти: **Покупець** і **Магазин**. До магазину мають ось-ось завезти новий товар, який цікавить покупця.

Покупець може щодня ходити до магазину, щоб перевіряти наявність товару. Але через це він буде дратуватися, даремно витрачаючи свій дорогоцінний час.



*Постійне відвідування магазину чи спам?*

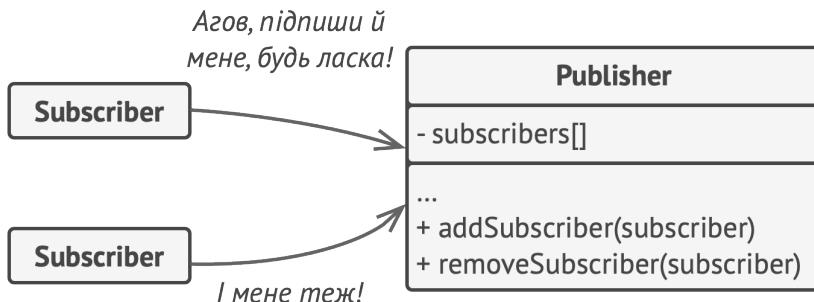
З іншого боку, магазин може розсылати спам кожному своєму покупцеві. Багатьох покупців це засмутить, оскільки товар специфічний і потрібний не всім.

Виходить конфлікт: або покупець гає час на періодичні перевірки, або магазин розтрачує ресурси на непотрібні сповіщення.

## 😊 Рішення

Давайте називати **видавцями** ті об'єкти, які містять важливий або цікавий для інших стан. Решту об'єктів, які хотіли б відстежувати зміни цього стану, назовемо **підписниками**.

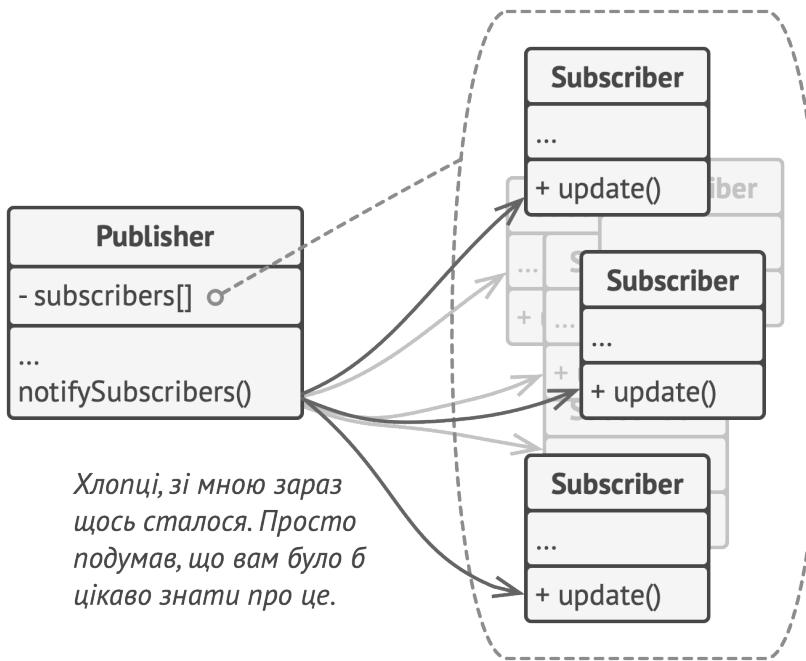
Патерн Спостерігач пропонує зберігати всередині об'єкта видавця список посилань на об'єкти підписників. Причому видавець не повинен вести список підписки самостійно. Він повинен надати методи, за допомогою яких підписники могли б додавати або прибирати себе зі списку.



*Підписка на події.*

Тепер найцікавіше. Коли у видавця відбуватиметься важлива подія, він буде проходитися за списком передплатників та сповіщувати їх про подію, викликаючи певний метод об'єктів-передплатників.

Видавцю байдуже, якого класу буде той чи інший підписник, бо всі вони повинні слідувати загальному інтерфейсу й мати єдиний метод оповіщення.



*Сповіщення про події.*

Побачивши, як добре все працює, ви можете виділити загальний інтерфейс і для всіх видавців, який буде складатися з методів підписки та відписки. Після цього підписники зможуть працювати з різними типами видавців, і отримувати від них сповіщення через єдиний метод.

## 🚘 Аналогія з життя

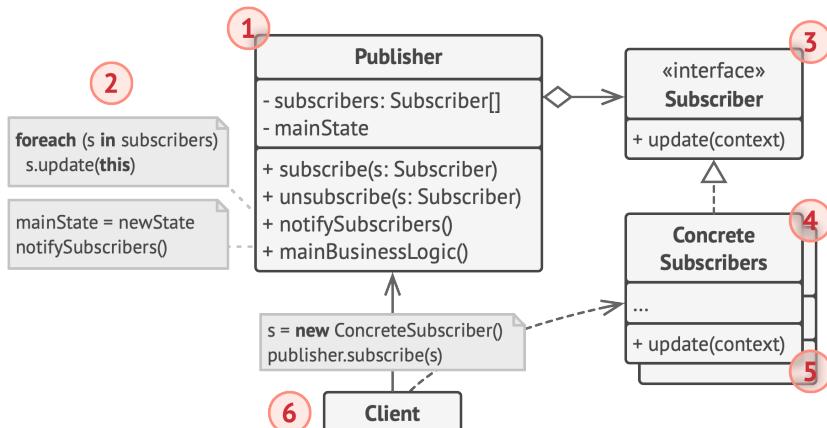
Після того, як ви оформили підписку на журнал, вам більше не потрібно їздити до супермаркета та дізнаватись, чи вже вийшов черговий номер. Натомість видавництво надсилає нові номери поштою прямо до вас додому, відразу після їхнього виходу.



*Передплата та доставка газет.*

Видавництво веде список підписників і знає, кому який журнал слати. Ви можете в будь-який момент відмовитися від підписки, ю журнал перестане до вас надходити.

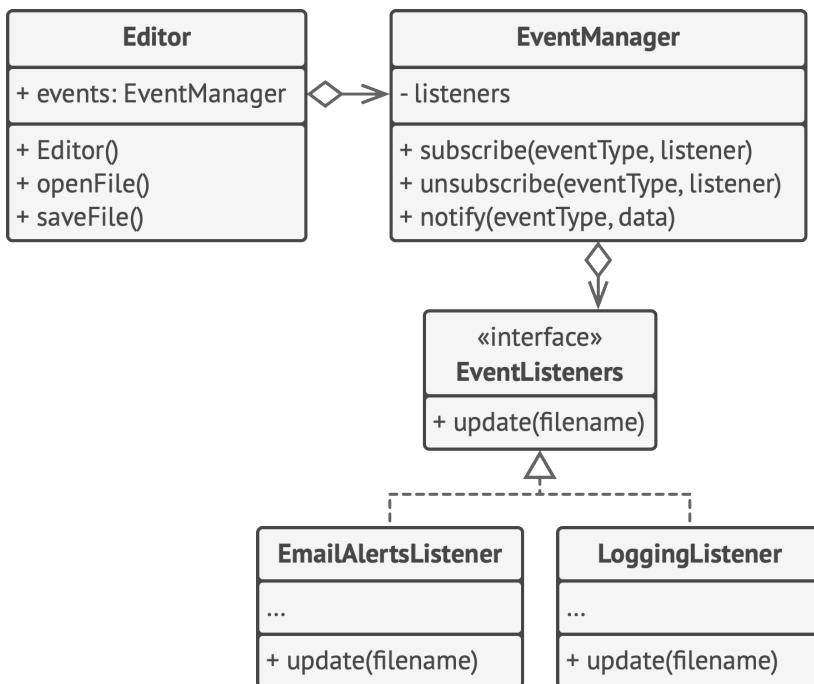
## Структура



1. **Видавець** володіє внутрішнім станом, зміни якого цікаво відслідковувати підписникам. Видавець містить механізм підписки: список підписників та методи підписки/відписки.
2. Коли внутрішній стан видавця змінюється, він сповіщає своїх підписників. Для цього видавець проходиться за списком підписників і викликає їхній метод сповіщення, який описаний в загальному інтерфейсі підписників.
3. **Підписник** визначає інтерфейс, яким користується видавець для надсилання сповіщень. Здебільшого для цього досить одного методу.
4. **Конкретні підписники** виконують щось у відповідь на сповіщення, яке надійшло від видавця. Ці класи мають дотримуватися загального інтерфейсу, щоб видавець не залежав від конкретних класів підписників.
5. Після отримання сповіщення підписнику необхідно отримати оновлений стан видавця. Видавець може передати цей стан через параметри методу сповіщення. Більш гнучкий варіант – передавати через параметри весь об'єкт видавця, щоб підписник міг сам отримати необхідні дані. Як варіант, підписник може постійно зберігати посилання на об'єкт видавця, переданий йому через конструктор.
6. **Клієнт** створює об'єкти видавців і підписників, а потім реєструє підписників на оновлення у видавцях.

## # Псевдокод

У цьому прикладі **Спостерігач** дає змогу об'єкту текстового редактора сповіщати інші об'єкти про зміни свого стану.



*Приклад сповіщення об'єктів про події в інших об'єктах.*

Список підписників складається динамічно, об'єкти можуть як підписуватися на певні події, так і відписуватися від них прямо під час виконання програми.

У цій реалізації редактор не веде список підписників самостійно, а делегує це вкладеному об'єкту. Це дає змогу вико-

ристовувати механізм підписки не лише в класі редактора, а і в інших класах програми.

Для додавання до програми нових підписників не потрібно змінювати класи видавців, допоки вони працюють із підписниками через загальний інтерфейс.

```

1 // Базовий клас-видавець. Містить код керування підписниками та
2 // надсилання їм сповіщень.
3 class EventManager is
4     private field listeners: hash map of event types and listeners
5
6     method subscribe(eventType, listener) is
7         listeners.add(eventType, listener)
8
9     method unsubscribe(eventType, listener) is
10        listeners.remove(eventType, listener)
11
12    method notify(eventType, data) is
13        foreach (listener in listeners.of(eventType)) do
14            listener.update(data)
15
16 // Конкретний клас-видавець, що містить цікаву для інших
17 // компонентів бізнес-логіку. Ми могли б зробити його прямим
18 // нащадком EventManager, але в реальному житті це не завжди є
19 // можливим (наприклад, якщо в класу вже є предок). Тому тут ми
20 // підключаємо механізм підписки за допомогою композиції.
21 class Editor is
22     public field events: EventManager
23     private field file: File
24

```

```
25  constructor Editor() is
26      events = new EventManager()
27
28  // Методи бізнес-логіки, які сповіщають підписників про
29  // зміни.
30  method openFile(path) is
31      this.file = new File(path)
32      events.notify("open", file.name)
33
34  method saveFile() is
35      file.write()
36      events.notify("save", file.name)
37  // ...
38
39
40 // Загальний інтерфейс підписників. У багатьох мовах, що мають
41 // функціональні типи, можна обйтися без цього інтерфейсу та
42 // конкретних класів, замінивши об'єкти підписників функціями.
43 interface EventListener is
44     method update(filename)
45
46
47 // Набір конкретних підписників. Кожен з них виконує якусь
48 // поведінку, реагуючи на сповіщення від видавця.
49 class LoggingListener implements EventListener is
50     private field log: File
51     private field message: string
52
53     constructor LoggingListener(log_filename, message) is
54         this.log = new File(log_filename)
55         this.message = message
56
```

```

57   method update(filename) is
58     log.write(replace('%s',filename,message))
59
60 class EmailAlertsListener implements EventListener is
61   private field email: string
62   private field message: string
63
64 constructor EmailAlertsListener(email, message) is
65   this.email = email
66   this.message = message
67
68 method update(filename) is
69   system.email(email, replace('%s',filename,message))
70
71
72 // Програма може сконфігурувати видавців та підписників, як
73 // завгодно, залежно від цілей та оточення.
74 class Application is
75   method config() is
76     editor = new Editor()
77
78     logger = new LoggingListener(
79       "/path/to/log.txt",
80       "Someone has opened file: %s");
81     editor.events.subscribe("open", logger)
82
83     emailAlerts = new EmailAlertsListener(
84       "admin@example.com",
85       "Someone has changed the file: %s")
86     editor.events.subscribe("save", emailAlerts)

```

## Застосування

-  Якщо після зміни стану одного об'єкта потрібно щось зробити в інших, але ви не знаєте наперед, які саме об'єкти мають відреагувати.
-  Описана проблема може виникнути при розробленні бібліотек користувачього інтерфейсу, якщо вам необхідно надати можливість стороннім класам реагувати на кліки по кнопках.

Патерн Спостерігач надає змогу будь-якому об'єкту з інтерфейсом підписника зареєструватися для отримання сповіщень про події, що трапляються в об'єктах-видавцях.

-  Якщо одні об'єкти мають спостерігати за іншими, але тільки у визначених випадках.
-  Видавці ведуть динамічні списки. Усі спостерігачі можуть підписуватися або відписуватися від отримання сповіщень безпосередньо під час виконання програми.

## Кроки реалізації

1. Розбийте вашу функціональність на дві частини: незалежне ядро та опціональні залежні частини. Незалежне ядро стане видавцем. Залежні частини стануть підписниками.

2. Створіть інтерфейс підписників. Зазвичай достатньо визнати в ньому лише один метод сповіщення.
3. Створіть інтерфейс видавців та опишіть у ньому операції керування підпискою. Пам'ятайте, що видавці повинні працювати з підписниками тільки через їхній загальний інтерфейс.
4. Вам потрібно вирішити, куди помістити код ведення підписки, адже він зазвичай буває однаковим для всіх типів видавців. Найочевидніший спосіб – це внесення коду до проміжного абстрактного класу, від якого будуть успадковуватися всі видавці.

Якщо ж ви інтегруєте патерн до існуючих класів, то створити новий базовий клас може бути важко. У цьому випадку ви можете помістити логіку підписки в допоміжний об'єкт та делегувати йому роботу з видавцями.

5. Створіть класи конкретних видавців. Реалізуйте їх таким чином, щоб після кожної зміні стану вони слали сповіщення всім своїм підписникам.
6. Реалізуйте метод сповіщення в конкретних підписниках. Не забудьте передбачити параметри, через які видавець міг би відправляти якісь дані, пов'язані з подією, що відбулась.

Можливий і інший варіант, коли підписник, отримавши сповіщення, сам візьме потрібні дані з об'єкта видавця. Але в

цьому разі ви будете змушені прив'язати клас підписника до конкретного класу видавця.

- Клієнт повинен створювати необхідну кількість об'єктів підписників та підписувати їх у видавців.

## **Переваги та недоліки**

- ✓ Видавці не залежать від конкретних класів підписників і навпаки.
- ✓ Ви можете підписувати і відписувати одержувачів «на льоту».
- ✓ Реалізує *принцип відкритості/закритості*.
- ✗ Підписники сповіщуються у випадковій послідовності.

## **Відносини з іншими патернами**

- Ланцюжок обов'язків, Команда Посередник та Спостерігач показують різні способи роботи тих, хто надсилає запити, та тих, хто їх отримує:
  - *Ланцюжок обов'язків* передає запит послідовно через ланцюжок потенційних отримувачів, очікуючи, що один з них обробить запит.
  - *Команда* встановлює непрямий односторонній зв'язок від відправників до одержувачів.

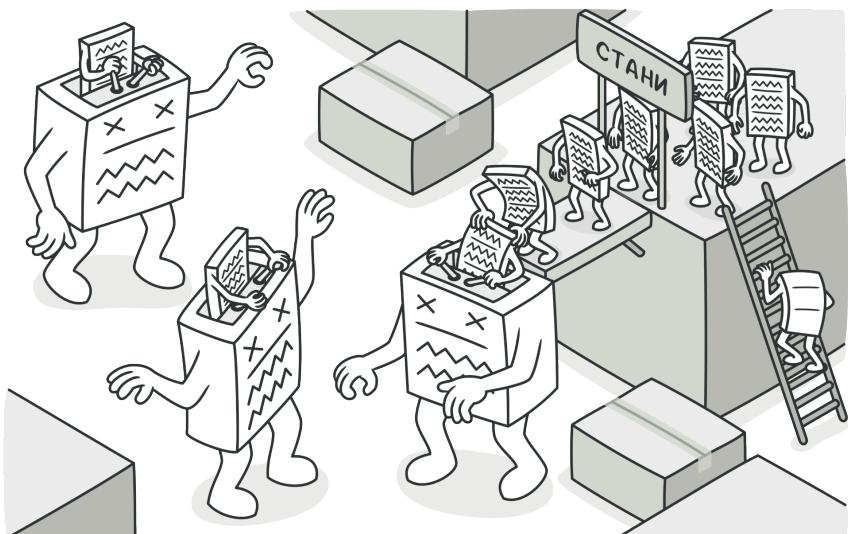
- *Посередник* прибирає прямий зв’язок між відправниками та одержувачами, змушуючи їх спілкуватися опосередковано, через себе.
- *Спостерігач* передає запит одночасно всім зацікавленим одержувачам, але дозволяє їм динамічно підписуватися або відписуватися від таких повідомлень.
- Різниця між **Посередником** та **Спостерігачем** не завжди очевидна. Найчастіше вони виступають як конкуренти, але іноді можуть працювати разом.

Мета *Посередника* – прибрati взаємні залежності між компонентами системи. Замість цього вони стають залежними від самого посередника. З іншого боку, мета *Спостерігача* – забезпечити динамічний односторонній зв’язок, в якому одні об’єкти опосередковано залежать від інших.

Досить популярною є реалізація *Посередника* за допомогою *Спостерігача*. При цьому об’єкт посередника буде виступати видавцем, а всі інші компоненти стануть передплатниками та зможуть динамічно стежити за подіями, що відбуваються у посереднику. У цьому випадку важко зрозуміти, чим саме відрізняються обидва патерни.

Але *Посередник* має й інші реалізації, коли окремі компоненти жорстко прив’язані до об’єкта посередника. Такий код навряд чи буде нагадувати *Спостерігача*, але залишиться *Посередником*.

Навпаки, у разі реалізації посередника з допомогою *Спостерігача*, представимо чи уявімо таку програму, в якій кожен компонент системи стає видавцем. Компоненти можуть підписуватися один на одного, не прив'язуючись до конкретних класів. Програма складатиметься з цілої мережі *Спостерігачів*, не маючи центрального об'єкта *Посередника*.



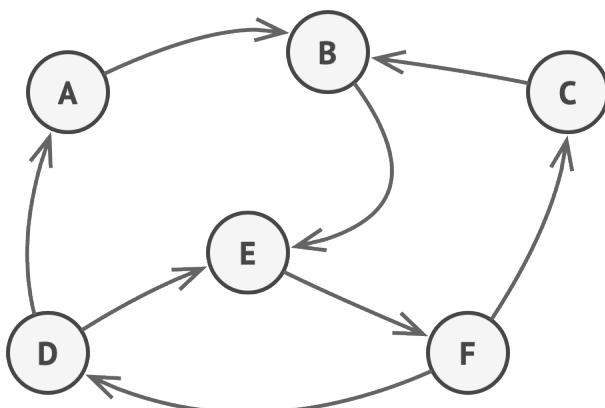
# СТАН

Також відомий як: *State*

**Стан** – це поведінковий патерн проєктування, що дає змогу об'єктам змінювати поведінку в залежності від їхнього стану. Ззовні створюється враження, ніби змінився клас об'єкта.

## :( Проблема

Патерн Стан неможливо розглядати у відриві від концепції **машини станів**, також відомої як *стейт-машина* або *кінцевий автомат*.



*Кінцевий автомат.*

Основна ідея в тому, що програма може знаходитися в одному з кількох станів, які увесь час змінюють один одного. Набір цих станів, а також переходів між ними, визначений наперед та *кінцевий*. Перебуваючи в різних станах, програма може по-різному реагувати на одні і ті самі події, що відбуваються з нею.

Такий підхід можна застосувати і до окремих об'єктів. Наприклад, об'єкт **Документ** може приймати три стани: **Чернетка**, **Модерація** або **Опублікований**. У кожному з цих станів метод **опублікувати** працюватиме по-різному:

- З чернетки він надішле документ на модерацію.
- З модерації – в публікацію, але за умови, що це зробив адміністратор.
- В опублікованому стані метод не буде робити нічого.



*Можливі стани документу та переходи між ними.*

Машину станів найчастіше реалізують за допомогою множини умовних операторів, `if` або `switch`, які перевіряють поточний стан об'єкта та виконують відповідну поведінку. Ймовірніше за все, ви вже реалізували у своєму житті хоча б одну машину станів, навіть не знаючи про це. Не вірите? Як щодо такого коду, виглядає знайомо?

```

1  class Document is
2      field state: string
3      // ...
4      method publish() is
5          switch (state)
6              "draft":
7                  state = "moderation"
8                  break
9              "moderation":
10                 if (currentUser.role == 'admin')
11                     state = "published"
12                     break
13                 "published":
14                     // Do nothing.
15                     break
16             // ...

```

Побудована таким чином машина станів має критичну ваду, яка покаже себе, якщо до Документа додати ще з десяток станів. Кожен метод буде складатися з об'ємного умовного оператора, який перебирає доступні стани.

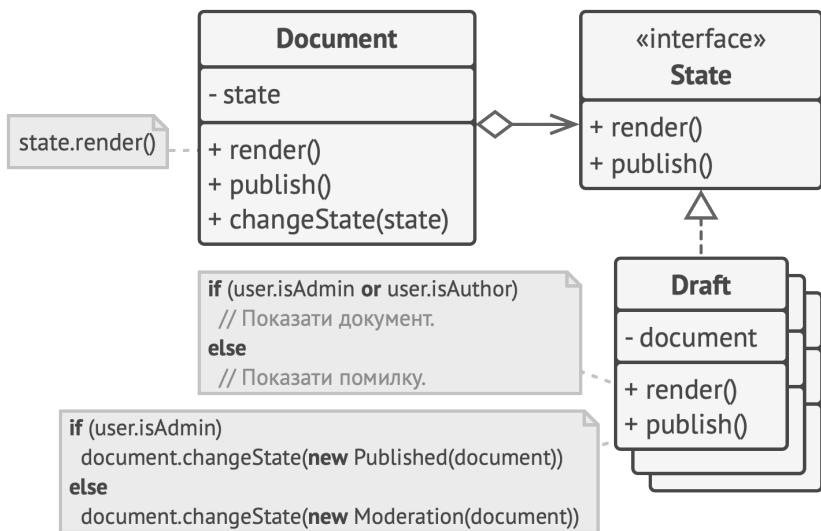
Такий код дуже складно підтримувати. Навіть найменша зміна логіки переходів змусить вас перевіряти роботу всіх методів, які містять умовні оператори машини станів.

Плутаниця та нагромадження умов особливо сильно проявляється в старих проектах. Набір можливих станів бував важко визначити заздалегідь, тому вони увесь час додаються в процесі еволюції програми. Через це рішення, що

здавалося простим і ефективним на початку розробки проекту, може згодом стати проекцією величезного макаронного монстра.

## 😊 Рішення

Патерн Стан пропонує створити окремі класи для кожного стану, в якому може перебувати контекстний об'єкт, а потім винести туди поведінки, що відповідають цим станам.



*Сторінка делегує виконання своєму активному стану.*

Замість того, щоб зберігати код всіх станів, початковий об'єкт, який звється контекстом, міститиме посилання на один з об'єктів-станів і делегуватиме йому роботу в залежності від стану.

Завдяки тому, що об'єкти станів матимуть спільний інтерфейс, контекст зможе делегувати роботу стану, не прив'язуючись до його класу. Поведінку контексту можна буде змінити в будь-який момент, підключивши до нього інший об'єкт-стан.

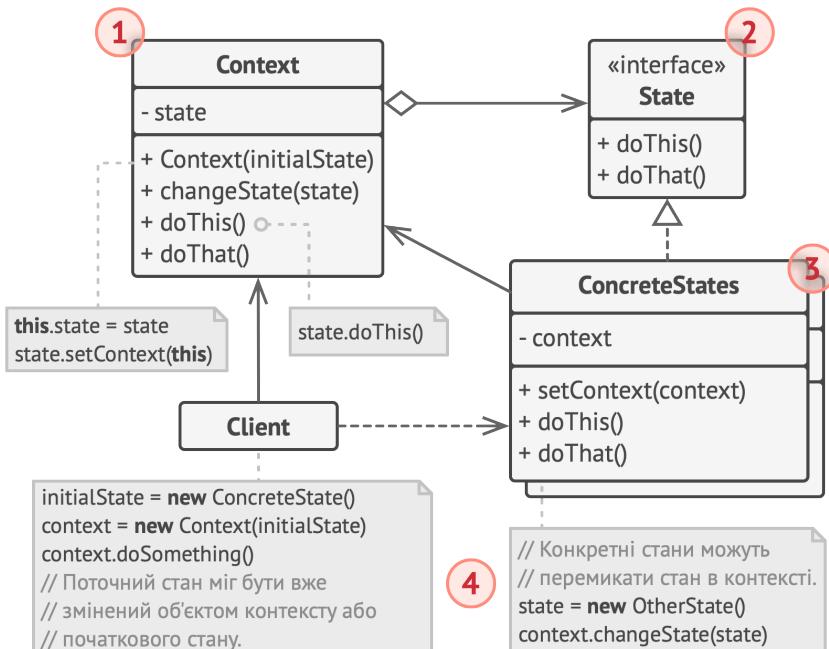
Дуже важливим нюансом, який відрізняє цей патерн від Стратегії, є те, що і контекст, і конкретні стани можуть знати один про одного та ініціювати переходи від одного стану до іншого.

## Аналогія з життя

Ваш смартфон поводиться по-різному в залежності від поточного стану:

- Якщо телефон розблоковано, натискання кнопок телефону призведе до якихось дій.
- Якщо телефон заблоковано, натискання кнопок призведе до появи екрану розблокування.
- Якщо телефон розряджено, натискання кнопок призведе до появи екрану зарядки.

## Структура

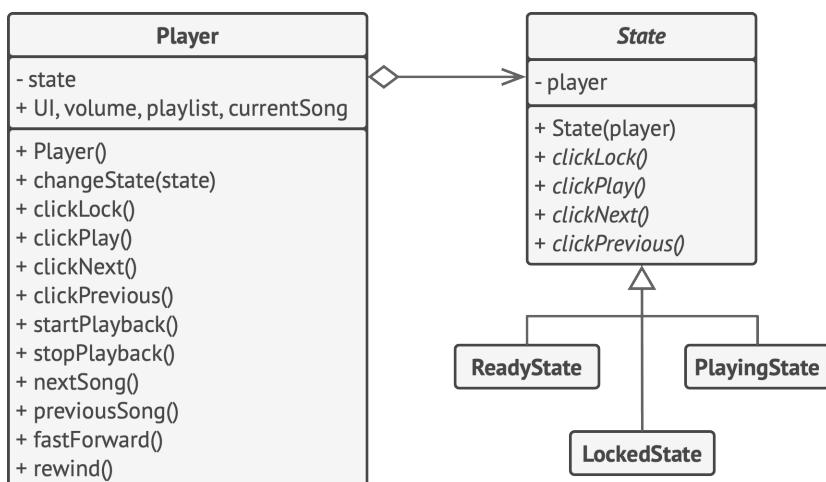


1. **Контекст** зберігає посилання на об'єкт стану та делегує йому частину роботи, яка залежить від станів. Контекст працює з цим об'єктом через загальний інтерфейс станів. Контекст повинен мати метод для присвоєння йому нового об'єкта-стану.
2. **Стан** описує спільний для всіх конкретних станів інтерфейс.
3. **Конкретні стани** реалізують поведінки, пов'язані з певним станом контексту. Іноді доводиться створювати цілі ієрархії класів станів, щоб узагальнити дублюючий код.

Стан може мати зворотнє посилання на об'єкт контексту. Через нього не тільки зручно отримувати з контексту потрібну інформацію, але й здійснювати зміну стану.

- І контекст, і об'єкти конкретних станів можуть вирішувати, коли і який стан буде обрано наступним. Щоб перемкнути стан, потрібно подати інший об'єкт-стан до контексту.

## # Псевдокод



*Приклад зміни поведінки програвача за допомогою станів.*

У цьому прикладі патерн **Стан** змінює функціональність одних і тих самих елементів керування музичним програвачем, залежно від стану, в якому зараз знаходиться програвач.

Об'єкт програвача містить об'єкт-стан, якому й делегує головну роботу. Змінюючи стан, можна впливати на те, як поводяться елементи керування програвача.

```

1 // Загальний інтерфейс усіх станів.
2 abstract class State is
3     protected field player: AudioPlayer
4
5     // Контекст передає себе до конструктора стану, щоб стан міг
6     // звертатися до його даних та методів у майбутньому, якщо
7     // буде потрібно.
8     constructor State(player) is
9         this.player = player
10
11    abstract method clickLock()
12    abstract method clickPlay()
13    abstract method clickNext()
14    abstract method clickPrevious()
15
16
17 // Конкретні стани реалізують методи загального стану по-своєму.
18 class LockedState extends State is
19
20     // При розблокуванні програвача із заблокованими клавішами,
21     // він може прийняти один з двох станів.
22     method clickLock() is
23         if (player.playing)
24             player.changeState(new PlayingState(player))
25         else
26             player.changeState(new ReadyState(player))
27

```

```
28 method clickPlay() is
29     // Нічого не робити.
30
31 method clickNext() is
32     // Нічого не робити.
33
34 method clickPrevious() is
35     // Нічого не робити.
36
37
38 // Конкретні стани самі можуть переводити контекст в інші стани.
39 class ReadyState extends State is
40     method clickLock() is
41         player.changeState(new LockedState(player))
42
43     method clickPlay() is
44         player.startPlayback()
45         player.changeState(new PlayingState(player))
46
47     method clickNext() is
48         player.nextSong()
49
50     method clickPrevious() is
51         player.previousSong()
52
53 class PlayingState extends State is
54     method clickLock() is
55         player.changeState(new LockedState(player))
56
57     method clickPlay() is
58         player.stopPlayback()
59         player.changeState(new ReadyState(player))
```

```
60 method clickNext() is
61     if (event.doubleclick)
62         player.nextSong()
63     else
64         player.fastForward(5)
65
66 method clickPrevious() is
67     if (event.doubleclick)
68         player.previous()
69     else
70         player.rewind(5)
71
72
73 // Програвач виступає в ролі контексту.
74 class AudioPlayer is
75     field state: State
76     field UI, volume, playlist, currentSong
77
78 constructor AudioPlayer() is
79     this.state = new ReadyState(this)
80
81     // Контекст змушує стан реагувати на користувачький ввід
82     // замість себе. Реакція може бути різною, залежно від
83     // того, який стан зараз активний.
84     UI = new UserInterface()
85     UI.lockButton.onClick(this.clickLock)
86     UI.playButton.onClick(this.clickPlay)
87     UI.nextButton.onClick(this.clickNext)
88     UI.prevButton.onClick(this.clickPrevious)
89
90     // Інші об'єкти теж повинні мати можливість замінити стан
91     // програвача.
```

```
92 method changeState(state: State) is
93     this.state = state
94
95 // Методи UI делегуватимуть роботу активному стану.
96 method clickLock() is
97     state.clickLock()
98 method clickPlay() is
99     state.clickPlay()
100 method clickNext() is
101     state.clickNext()
102 method clickPrevious() is
103     state.clickPrevious()
104
105 // Сервісні методи контексту, що викликаються станами.
106 method startPlayback() is
107     ...
108 method stopPlayback() is
109     ...
110 method nextSong() is
111     ...
112 method previousSong() is
113     ...
114 method fastForward(time) is
115     ...
116 method rewind(time) is
117     ...
```

## Застосування

-  Якщо у вас є об'єкт, поведінка якого кардинально змінюється в залежності від внутрішнього стану, причому типів станів багато, а іхній код часто змінюється.
-  Патерн пропонує виділити в окремі класи всі поля й методи, пов'язані з визначеним станом. Початковий об'єкт буде постійно посилатися на один з об'єктів-станів, делегуючи йому частину своєї роботи. Для зміни стану до контексту достатньо буде підставляти інший об'єкт-стан.
-  Якщо код класу містить безліч великих, схожих один на одного умовних операторів, які вибирають поведінки в залежності від поточних значень полів класу.
-  Патерн пропонує перемістити кожну гілку такого умовного оператора до власного класу. Сюди ж можна поселити й усі поля, пов'язані з цим станом.
-  Якщо ви свідомо використовуєте табличну машину станів, побудовану на умовних операторах, але змушені миритися з дублюванням коду для схожих станів та переходів.
-  Патерн Стан дозволяє реалізувати ієрархічну машину станів, що базується на наслідуванні. Ви можете успадкувати схожі стани від одного батьківського класу та винести туди весь дублюючий код.

## Кроки реалізації

1. Визначтеся з класом, який відіграватиме роль контексту. Це може бути як існуючий клас, який вже має залежність від стану, так і новий клас, якщо код станів «розмазаний» по кількох класах.
2. Створіть загальний інтерфейс станів. Він повинен описувати методи, спільні для всіх станів, виявлених у контексті. Зверніть увагу, що не всю поведінку контексту потрібно переносити до стану, а тільки ту, яка залежить від станів.
3. Для кожного фактичного стану створіть клас, який реалізує інтерфейс стану. Перемістіть код, пов'язаний з конкретними станами, до потрібних класів. Зрештою, всі методи інтерфейсу стану повинні бути реалізовані в усіх класах станів.

При перенесенні поведінки з контексту ви можете зіткнутися з тим, що ця поведінка залежить від приватних полів або методів контексту, до яких немає доступу з об'єкта стану. Є кілька способів, щоб обійти цю проблему.

Найпростіший – залишити поведінку всередині контексту, викликаючи його з об'єкта стану. З іншого боку, ви можете зробити класи станів вкладеними до класу контексту, і тоді вони отримають доступ до всіх приватних частин контексту. Останній спосіб, щоправда, доступний лише в деяких мовах програмування (наприклад, Java, C#).

4. Створіть в контексті поле для зберігання об'єктів-станів, а також публічний метод для зміни значення цього поля.
5. Старі методи контексту, в яких перебував залежний від стану код, замініть на виклики відповідних методів об'єкта-стану.
6. В залежності від бізнес-логіки, розмістіть код, який перемікає стан контексту, або всередині контексту, або всередині класів конкретних станів.

## **Переваги та недоліки**

- ✓ Позбавляє від безлічі великих умовних операторів машини станів.
- ✓ Концентрує в одному місці код, пов'язаний з певним станом.
- ✓ Спрощує код контексту.
- ✗ Може невіправдано ускладнити код, якщо станів мало, і вони рідко змінюються.

## **Відносини з іншими патернами**

- **Міст, Стратегія та Стан** (а також трохи і **Адаптер**) мають схожі структури класів – усі вони побудовані за принципом «композиції», тобто делегування роботи іншим об'єктам. Проте вони відрізняються тим, що вирішують різні проблеми. Пам'ятайте, що патерни – це не тільки рецепт побудови коду певним чином, але й описування проблем, які призвели до такого рішення.

- **Стан** можна розглядати як надбудову над **Стратегією**. Обидва патерни використовують композицію, щоб змінювати поведінку головного об'єкта, делегуючи роботу вкладеним об'єктам-помічникам. Проте в *Стратегії* ці об'єкти не знають один про одного і жодним чином не пов'язані. У *Стані* конкретні стани самостійно можуть перемікати контекст.



# СТРАТЕГІЯ

Також відомий як: *Strategy*

**Стратегія** – це поведінковий патерн проектування, який визначає сімейство схожих алгоритмів і розміщує кожен з них у власному класі. Після цього алгоритми можна замінити один на інший прямо під час виконання програми.

## Проблема

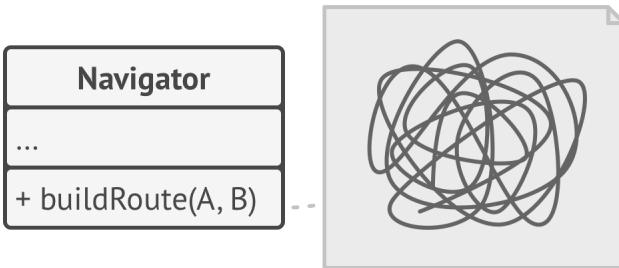
Ви вирішили написати програму-навігатор для подорожуючих. Вона повинна показувати гарну й зручну карту, яка дозволяла б з легкістю орієнтуватися в незнайомому місті.

Однією з найбільш очікуваних функцій був пошук та прокладання маршрутів. Перебуваючи в невідомому йому місті, користувач повинен мати можливість вказати початкову точку та пункт призначення, а навігатор, в свою чергу, прокладе оптимальний шлях.

Перша версія вашого навігатора могла прокладати маршрут лише автомобільними шляхами, тому чудово підходила для подорожей автомобілем. Але, вочевидь, не всі їздять у відпустку автомобілями. Тому наступним кроком ви додали до навігатора можливість прокладання піших маршрутів.

Через деякий час з'ясувалося, що частина туристів під час пересування містом віддають перевагу громадському транспорту. Тому ви додали ще й таку опцію прокладання шляху.

Але й це ще не все. У найближчій перспективі ви хотіли б додати прокладку маршрутів велодоріжками, а у віддаленому майбутньому – маршрути, пов’язані з відвідуванням цікавих та визначних місць.



*Код навігатора стає занадто роздутим.*

Якщо з популярністю навігатора не було жодних проблем, то технічна частина викликала запитання й періодичний головний біль. З кожним новим алгоритмом код основного класу навігатора збільшувався вдвічі. В такому великому класі стало важкувато орієнтуватися.

Будь-яка зміна алгоритмів пошуку, чи то виправлення багів, чи додавання нового алгоритму, зачіпала основний клас. Це підвищувало ризик створення помилки шляхом випадкового внесення змін до робочого коду.

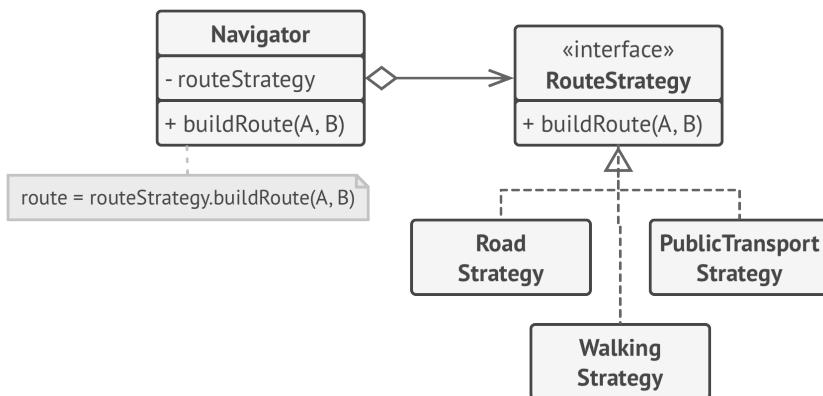
Крім того, ускладнювалася командна робота з іншими програмістами, яких ви найняли після успішного релізу навігатора. Ваші зміни нерідко торкалися одного і того самого коду, створюючи конфлікти, які вимагали додаткового часу на їхнє вирішення.

## 😊 Рішення

Патерн Стратегія пропонує визначити сімейство схожих алгоритмів, які часто змінюються або розширяються, й винести їх до власних класів, які називають *стратегіями*.

Замість того, щоб початковий клас сам виконував той чи інший алгоритм, він відіграватиме роль контексту, посилаючись на одну зі стратегій та делегуючи їй виконання роботи. Щоб змінити алгоритм, вам буде достатньо підставити в контекст інший об'єкт-стратегію.

Важливо, щоб всі стратегії мали єдиний інтерфейс. Використовуючи цей інтерфейс, контекст буде незалежним від конкретних класів стратегій. З іншого боку, ви зможете змінювати та додавати нові види алгоритмів, не чіпаючи код контексту.

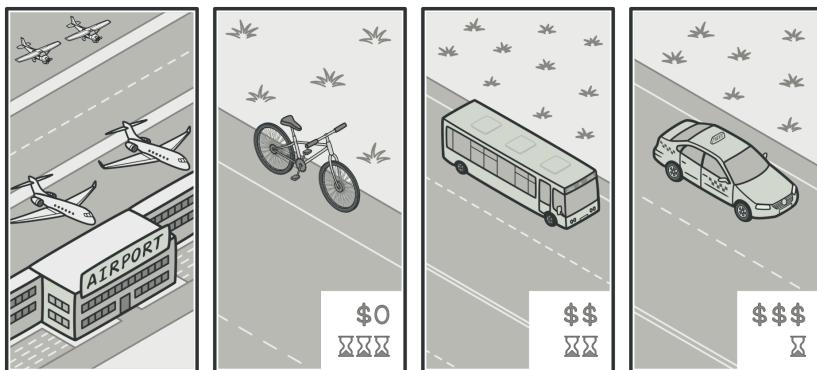


*Стратегії побудови шляху.*

У нашому прикладі кожен алгоритм пошуку шляху переїде до свого власного класу. В цих класах буде визначено лише один метод, що приймає в параметрах координати початку та кінця маршруту, а повертає масив всіх точок маршруту.

Хоча кожен клас прокладатиме маршрут на свій розсуд, для навігатора це не буде мати жодного значення, оскільки його робота полягає тільки у зображені маршруту. Навігатору достатньо подати до стратегії дані про початок та кінець маршруту, щоб отримати масив точок маршруту в обумовленому форматі. Клас навігатора буде мати метод для встановлення стратегії, що дозволить змінювати стратегію пошуку шляху «на льоту». Цей метод стане у нагоді клієнтському коду навігатора, наприклад, кнопкам-перемикачам типів маршрутів в інтерфейсі користувача.

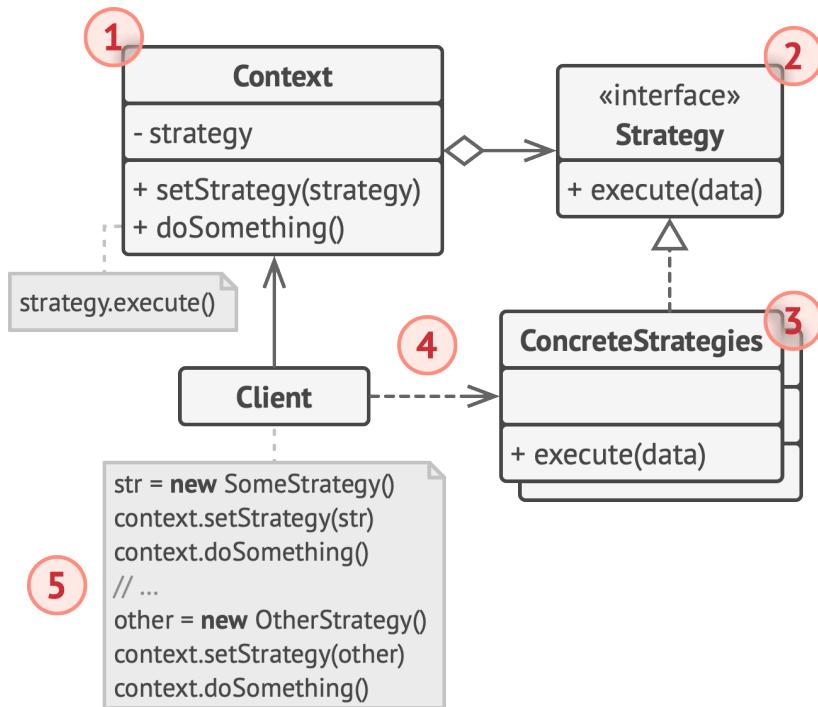
## Аналогія з життя



Різні стратегії потрапляння до аеропорту.

Вам потрібно дістатися аеропорту. Можна доїхати автобусом, таксі або велосипедом. Тут вид транспорту є стратегією. Ви вибираєте конкретну стратегію в залежності від контексту – наявності грошей або часу до відльоту.

## Структура



1. **Контекст** зберігає посилання на об'єкт конкретної стратегії, працюючи з ним через загальний інтерфейс стратегій.
2. **Стратегія** визначає інтерфейс, спільний для всіх варіацій алгоритму. Контекст використовує цей інтерфейс для виклику алгоритму.

Для контексту неважливо, яка саме варіація алгоритму буде обрана, оскільки всі вони мають одинаковий інтерфейс.

3. **Конкретні стратегії** реалізують різні варіації алгоритму.
4. Під час виконання програми контекст отримує виклики від клієнта й делегує їх об'єкту конкретної стратегії.
5. Клієнт повинен створити об'єкт конкретної стратегії та передати його до конструктора контексту. Крім того, клієнт повинен мати можливість замінити стратегію на льоту, використовуючи сетер поля стратегії. Завдяки цьому, контекст не знатиме про те, яку саме стратегію зараз обрано.

## # Псевдокод

У цьому прикладі контекст використовує **Стратегію** для виконання тієї чи іншої арифметичної операції.

```

1 // Загальний інтерфейс стратегій.
2 interface Strategy is
3     method execute(a, b)
4
5 // Кожна конкретна стратегія реалізує загальний інтерфейс у свій
6 // власний спосіб.
7 class ConcreteStrategyAdd implements Strategy is
8     method execute(a, b) is
9         return a + b
10

```

```
11 class ConcreteStrategySubtract implements Strategy is
12     method execute(a, b) is
13         return a - b
14
15 class ConcreteStrategyMultiply implements Strategy is
16     method execute(a, b) is
17         return a * b
18
19 // Контекст завжди працює зі стратегіями через загальний
20 // інтерфейс. Він не знає, яку саме стратегію йому подано.
21 class Context is
22     private strategy: Strategy
23
24     method setStrategy(Strategy strategy) is
25         this.strategy = strategy
26
27     method executeStrategy(int a, int b) is
28         return strategy.execute(a, b)
29
30
31 // Конкретна стратегія вибирається на більш високому рівні,
32 // наприклад, конфігуратором всієї програми. Готовий об'єкт-
33 // стратегія подається до клієнтського об'єкта, а потім може
34 // бути замінений іншою стратегією, в будь-який момент, «на
35 // льоту».
36 class ExampleApplication is
37     method main() is
38         // 1. Створити об'єкт контексту.
39         // 2. Ввести перше число (n1).
40         // 3. Ввести друге число (n2).
41         // 4. Ввести бажану операцію.
42         // 5. Потім, обрати стратегію:
```

```

43     if (action == addition) then
44         context.setStrategy(new ConcreteStrategyAdd())
45
46     if (action == subtraction) then
47         context.setStrategy(new ConcreteStrategySubtract())
48
49     if (action == multiplication) then
50         context.setStrategy(new ConcreteStrategyMultiply())
51
52     // 6. Виконати операцію за допомогою стратегії:
53     result = context.executeStrategy(n1, n2)
54
55     // N. Вивести результат на екран.

```

## Застосування

-  Якщо вам потрібно використовувати різні варіації якого-небудь алгоритму всередині одного об'єкта.
-  Стратегія дозволяє варіювати поведінку об'єкта під час виконання програми, підставляючи до нього різні об'єкти-поведінки (наприклад, що відрізняються балансом швидкості та споживання ресурсів).
-  Якщо у вас є безліч схожих класів, які відрізняються лише деякою поведінкою.

-  Стратегія дозволяє відокремити поведінку, що відрізняється, у власну ієрархію класів, а потім звести початкові класи до одного, налаштовуючи його поведінку стратегіями.
  
-  Якщо ви не хочете оголювати деталі реалізації алгоритмів для інших класів.
  
-  Стратегія дозволяє ізолювати код, дані й залежності алгоритмів від інших об'єктів, приховавши ці деталі всередині класів-стратегій.
  
-  Якщо різні варіації алгоритмів реалізовано у вигляді розлогого умовного оператора. Кожна гілка такого оператора є варіацією алгоритму.
  
-  Стратегія розміщує кожну лапу такого оператора до окремого класу-стратегії. Потім контекст отримує певний об'єкт-стратегію від клієнта й делегує йому роботу. Якщо раптом знадобиться змінити алгоритм, до контексту можна подати іншу стратегію.

## Кроки реалізації

1. Визначте алгоритм, що склонний до частих змін. Також підійде алгоритм, який має декілька варіацій, які обираються під час виконання програми.

2. Створіть інтерфейс стратегій, що описує цей алгоритм. Він повинен бути спільним для всіх варіантів алгоритму.
3. Помістіть варіації алгоритму до власних класів, які реалізують цей інтерфейс.
4. У класі контексту створіть поле для зберігання посилання на поточний об'єкт-стратегію, а також метод для її зміни. Переоконайтесь в тому, що контекст працює з цим об'єктом тільки через загальний інтерфейс стратегій.
5. Клієнти контексту мають подавати до нього відповідний об'єкт-стратегію, коли хочуть, щоб контекст поводився певним чином.

## Переваги та недоліки

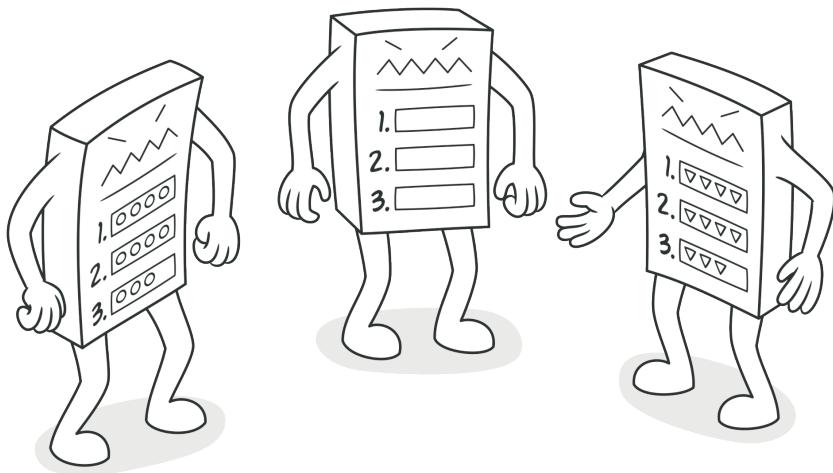
- ✓ Гаряча заміна алгоритмів на льоту.
- ✓ Ізоляє код і дані алгоритмів від інших класів.
- ✓ Заміна спадкування делегуванням.
- ✓ Реалізує *принцип відкритості/закритості*.
- ✗ Ускладнює програму внаслідок додаткових класів.
- ✗ Клієнт повинен знати, в чому полягає різниця між стратегіями, щоб вибрати потрібну.

## ↔ Відносини з іншими патернами

- **Міст, Стратегія та Стан** (а також трохи і **Адаптер**) мають схожі структури класів – усі вони побудовані за принципом «композиції», тобто делегування роботи іншим об'єктам. Проте вони відрізняються тим, що вирішують різні проблеми. Пам'ятайте, що патерни – це не тільки рецепт побудови коду певним чином, але й описування проблем, які привели до такого рішення.
- **Команда** та **Стратегія** схожі за принципом, але відрізняються масштабом та застосуванням:
  - *Команду* використовують для перетворення будь-яких різномірних дій на об'єкти. Параметри операції перетворюються на поля об'єкта. Цей об'єкт тепер можна логувати, зберігати в історії для скасування, передавати у зовнішні сервіси тощо.
  - З іншого боку, *Стратегія* описує різні способи того, як зробити одну і ту саму дію, дозволяючи замінювати ці способи в якомусь об'єкті контексту прямо під час виконання програми.
- **Стратегія** змінює поведінку об'єкта «зсередини», а **Декоратор** змінює його «ззовні».
- **Шаблонний метод** використовує спадкування, щоб розширявати частини алгоритму. **Стратегія** використовує делегування, щоб змінювати «на льоту» алгоритми, що

виконуються. *Шаблонний метод* працює на рівні класів. *Стратегія* дозволяє змінювати логіку окремих об'єктів.

- Стан можна розглядати як надбудову над Стратегією. Обидва патерни використовують композицію, щоб змінювати поведінку головного об'єкта, делегуючи роботу вкладеним об'єктам-помічникам. Проте в *Стратегії* ці об'єкти не знають один про одного і жодним чином не пов'язані. У *Стані* конкретні стани самостійно можуть перемикати контекст.



# ШАБЛОННИЙ МЕТОД

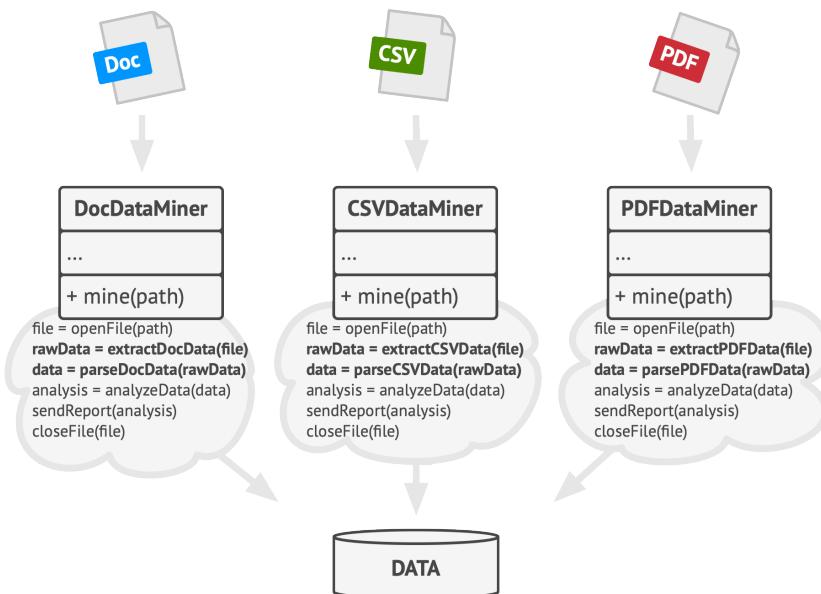
Також відомий як: *Template Method*

**Шаблонний метод** – це поведінковий патерн проектування, який визначає кістяк алгоритму, перекладаючи відповідальність за деякі його кроки на підкласи. Патерн дозволяє підкласам перевизначати кроки алгоритму, не змінюючи його загальної структури.

## :( Проблема

Ви пишете програму для дата-майнінгу в офісних документах. Користувачі завантажуватимуть до неї документи різних форматів (PDF, DOC, CSV), а програма повинна видобути з них корисну інформацію.

У першій версії ви обмежилися обробкою тільки DOC файлів. У наступній версії додали підтримку CSV. А через місяць «прикутили» роботу з PDF документами.



*Класи дата-майнінгу містять багато дублювань.*

В якийсь момент ви помітили, що код усіх трьох класів обробки документів хоч і відрізняється в частині роботи з файлами, але містить досить багато спільного в частині

ні самого видобування даних. Було б добре позбутися від повторної реалізації алгоритму видобування даних у кожному з класів.

До того ж інший код, який працює з об'єктами цих класів, наповнений умовами, що перевіряють тип обробника перед початком роботи. Весь цей код можна спростити, якщо злити всі три класи в одне ціле або звести їх до загального інтерфейсу.

## Рішення

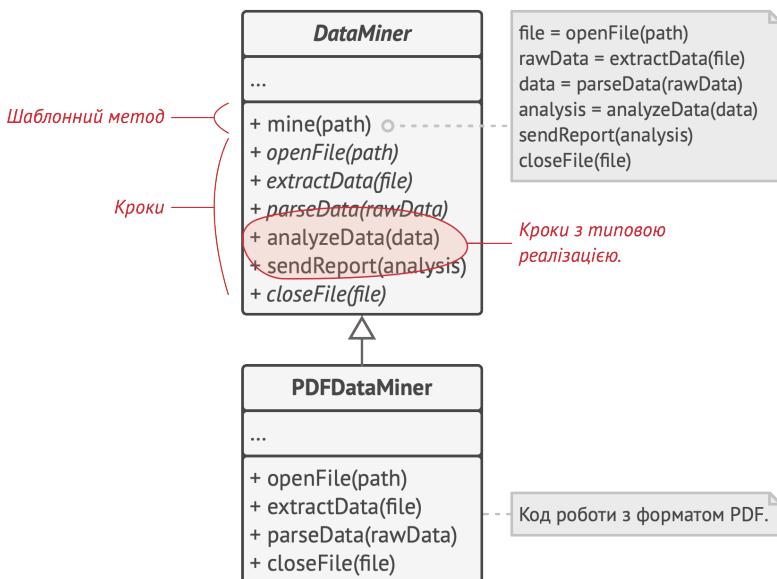
Патерн Шаблонний метод пропонує розбити алгоритм на послідовність кроків, описати ці кроки в окремих методах і викликати їх в одному *шаблонному* методі один за одним.

Це дозволить підкласам перевизначити деякі кроки алгоритму, залишаючи без змін його структуру та інші кроки, які для цього підкласу не є важливими.

У нашому прикладі з дата-майнінгом ми можемо створити загальний базовий клас для всіх трьох алгоритмів. Цей клас складатиметься з шаблонного методу, який послідовно викликає кроки розбору документів.

Для початку кроки шаблонного методу можна зробити абстрактними. З цієї причини усі підкласи повинні будуть реалізувати кожен з кроків по-своєму. В нашему випадку

всі підкласи вже містять реалізацію кожного з кроків, тому додатково нічого робити не потрібно.



Шаблонний метод розбиває алгоритм на кроки, дозволяючи підкласами перевизначити деякі з них.

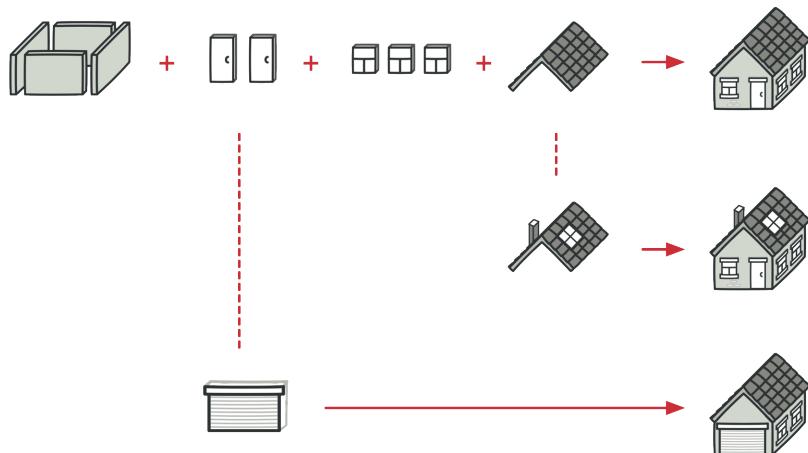
Справді важливим є наступний етап. Тепер ми можемо визначити спільну поведінку для всіх трьох класів і винести її до суперкласу. У нашому прикладі кроки відкривання та закривання документів відрізняються для всіх підкласів, тому залишається абстрактними. З іншого боку, код обробки даних, одинаковий для всіх типів документів, переїде до базового класу.

Як бачите, у нас з'явилося два типа кроків: *абстрактні*, що кожен підклас обов'язково має реалізувати, а також кроки з

*типовуою реалізацією, які можна перевизначити в підкласах, але це не обов'язково.*

Але є ще й третій тип кроків – хуки. Це опціональні кроки, які виглядають як звичайні методи, але взагалі не містять коду. Шаблонний метод залишиться робочим, навіть якщо жоден підклас не перевизначить такий хук. Підсумовуючи сказане, хук дає підкласам додаткові точки «вклинування» в хід шаблонного методу.

## 🚗 Аналогія з життя



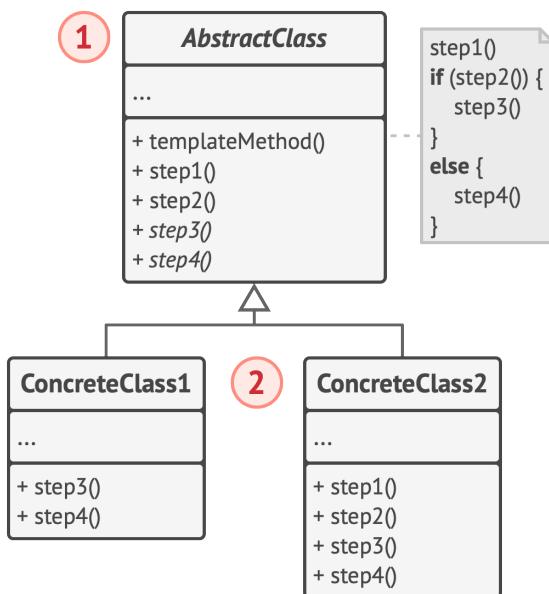
*Проект типового будинку можуть трохи змінити за бажанням клієнта.*

Під час будівництва типових будинків будівельники використовують підхід, схожий на шаблонний метод. У них є основний архітектурний проект, в якому розписані кроки

будівництва: заливка фундаменту, витягування стін, покриття даху, встановлення вікон тощо.

Але, незважаючи на стандартизацію кожного етапу, будівельники можуть робити невеликі зміни на кожному з етапів, щоб зробити будинок трішечки не схожим на інші.

## Структура

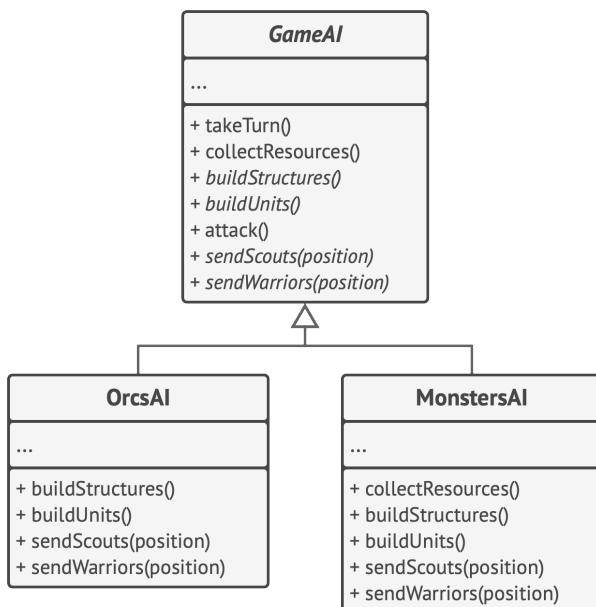


1. **Абстрактний клас** визначає кроки алгоритму й містить шаблонний метод, що складається з викликів цих кроків. Кроки можуть бути як абстрактними, так і містити реалізацію за замовчуванням.

2. **Конкретний клас** перевизначає деякі або всі кроки алгоритму. Конкретні класи не перевизначають сам шаблонний метод.

## # Псевдокод

У цьому прикладі **Шаблонний метод** використовується як заготовка для стандартного штучного інтелекту в простій стратегічній грі. Для введення в гру нової раси достатньо створити підклас і реалізувати в ньому відсутні методи.



*Приклад класів штучного інтелекту для простої гри.*

Всі раси гри матимуть приблизно однакові типи юнітів та будівель, тому структура штучного інтелекту буде однаковою. Але різні раси можуть різним шляхом реалізувати ці

кроки. Так, наприклад, орки будуть агресивнішими в атаці, люди більш активними в захисті, а дикі монстри взагалі не будуть займатися будівництвом.

```

1  class GameAI is
2      // Шаблонний метод повинен бути заданий у базовому класі.
3      // Він складається з викликів методів у певному порядку.
4      // Здебільшого, ці методи є кроками якогось алгоритму.
5      method turn() is
6          collectResources()
7          buildStructures()
8          buildUnits()
9          attack()
10
11     // Деякі з цих методів можуть бути реалізовані безпосередньо
12     // у базовому класі.
13     method collectResources() is
14         foreach (s in this.builtStructures) do
15             s.collect()
16
17     // А деякі можуть бути повністю абстрактними.
18     abstract method buildStructures()
19     abstract method buildUnits()
20
21     // До речі, клас може мати більше одного шаблонного методу.
22     method attack() is
23         enemy = closestEnemy()
24         if (enemy == null)
25             sendScouts(map.center)
26         else
27             sendWarriors(enemy.position)

```

```
28     abstract method sendScouts(position)
29     abstract method sendWarriors(position)
30
31 // Підкласи можуть надавати свою реалізацію кроків алгоритму, не
32 // змінюючи сам шаблонний метод.
33 class OrcsAI extends GameAI is
34     method buildStructures() is
35         if (there are some resources) then
36             // Будувати ферми, потім бараки, а потім цитадель.
37
38     method buildUnits() is
39         if (there are plenty of resources) then
40             if (there are no scouts)
41                 // Побудувати раба, додати до групи розвідників.
42         else
43             // Побудувати піхотинця, додати до групи воїнів.
44
45     // ...
46
47     method sendScouts(position) is
48         if (scouts.length > 0) then
49             // Відправити розвідників на позицію.
50
51     method sendWarriors(position) is
52         if (warriors.length > 5) then
53             // Відправити воїнів на позицію.
54
55 // Підкласи можуть не тільки реалізовувати абстрактні крохи, але
56 // й перевизначати крохи, вже реалізовані в базовому класі.
57 class MonstersAI extends GameAI is
58     method collectResources() is
59         // Нічого не робити.
```

```

60 method buildStructures() is
61     // Нічого не робити.
62
63 method buildUnits() is
64     // Нічого не робити.

```

## Застосування

-  Якщо підкласи повинні розширювати базовий алгоритм, не змінюючи його структури.
-  Шаблонний метод дозволяє підкласами розширювати певні кроки алгоритму через спадкування, не змінюючи при цьому структуру алгоритмів, оголошеної в базовому класі.
-  Якщо у вас є кілька класів, які роблять одне й те саме з незначними відмінностями. Якщо ви редагуєте один клас, тоді доводиться вносити такі ж виправлення до інших класів.
-  Патерн шаблонний метод пропонує створити для схожих класів спільний суперклас та оформити в ньому головний алгоритм у вигляді кроків. Кроки, які відрізняються, можна перевизначити у підкласах.

Це дозволить прибрати дублювання коду в кількох класах, які відрізняються деталями, але мають схожу поведінку.

## Кроки реалізації

1. Вивчіть алгоритм і подумайте, чи можна його розбити на кроки. Вирішіть, які кроки будуть стандартними для всіх варіацій алгоритму, а які можуть бути змінюваними.
2. Створіть абстрактний базовий клас. Визначте в ньому шаблонний метод. Цей метод повинен складатися з викликів кроків алгоритму. Є сенс у тому, щоб зробити шаблонний метод фінальним, аби підкласи не могли перевизначити його (якщо ваша мова програмування це дозволяє).
3. Додайте до абстрактного класу методи для кожного з кроків алгоритму. Ви можете зробити ці методи абстрактними або додати якусь типову реалізацію. У першому випадку всі підкласи *повинні* будуть реалізувати ці методи, а в другому – тільки якщо реалізація кроку в підкласі відрізняється від стандартної версії.
4. Подумайте про введення хуків в алгоритм. Найчастіше хуки розташовують між основними кроками алгоритму, а також до та після всіх кроків.
5. Створіть конкретні класи, успадкувавши їх від абстрактного класу. Реалізуйте в них всі кроки та хуки, яких не вистачає.

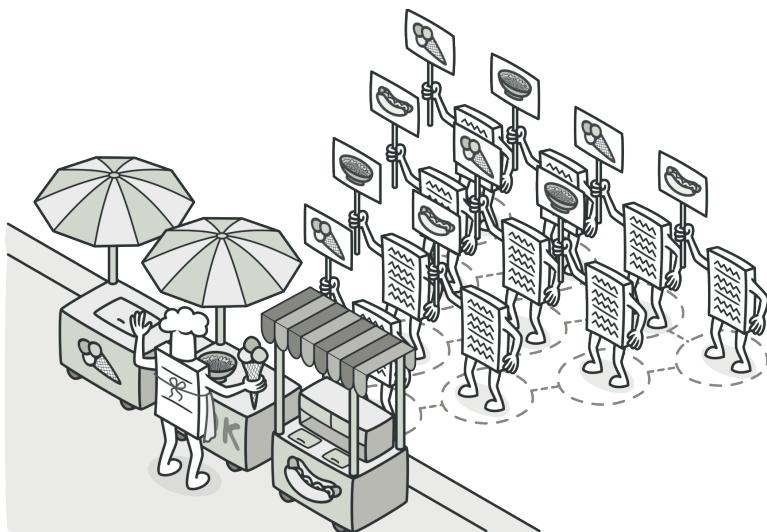
## Переваги та недоліки

- ✓ Полегшує повторне використання коду.

- ✗ Ви жорстко обмежені скелетом існуючого алгоритму.
- ✗ Ви можете порушити *принцип підстановки Барбари Лісков*, змінюючи базову поведінку одного з кроків алгоритму через підклас.
- ✗ У міру зростання кількості кроків шаблонний метод стає занадто складно підтримувати.

## ↔ Відносини з іншими патернами

- Фабричний метод можна розглядати як окремий випадок Шаблонного методу. Крім того, Фабричний метод нерідко буває частиною великого класу з Шаблонними методами.
- Шаблонний метод використовує спадкування, щоб розширявати частини алгоритму. Стратегія використовує делегування, щоб змінювати «на льоту» алгоритми, що виконуються. Шаблонний метод працює на рівні класів. Стратегія дозволяє змінювати логіку окремих об'єктів.



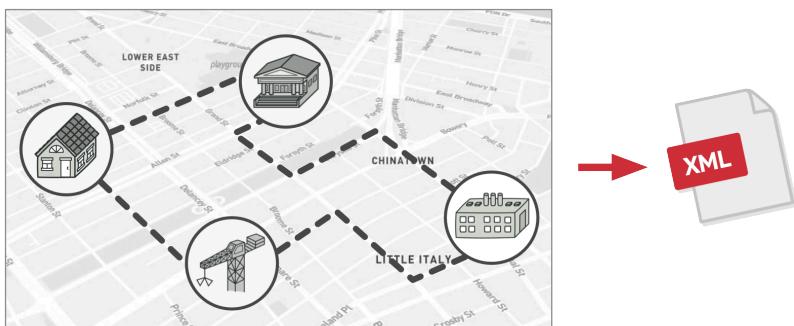
# ВІДВІДУВАЧ

Також відомий як: *Visitor*

**Відвідувач** – це поведінковий патерн проектування, що дає змогу додавати до програми нові операції, не змінюючи класи об'єктів, над якими ці операції можуть виконуватися.

## (:() Проблема

Ваша команда розробляє програму, що працює з геоданими у вигляді графа. Вузлами графа можуть бути як міста, так інші локації, такі, як пам'ятки, великих підприємств тощо. Кожен вузол має посилання на найближчі до нього вузли. Для кожного типу вузла існує свій власний клас, а кожен вузол представлений окремим об'єктом.

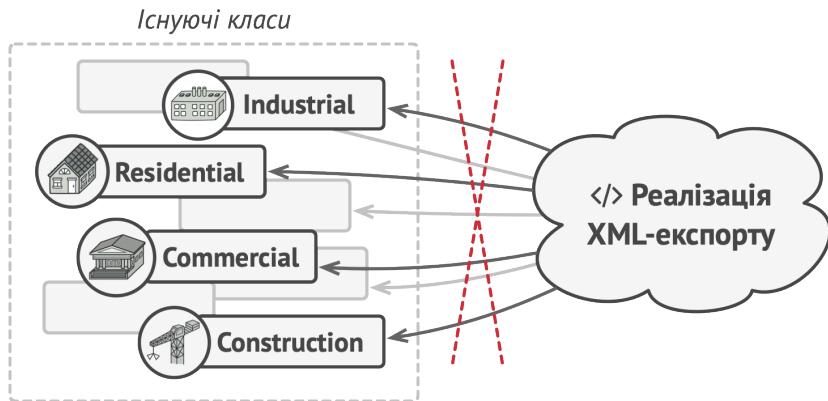


*Експорт гео-вузлів до XML.*

Ваше завдання – зробити експорт цього графа до XML. Справа була б легкою, якщо б ви могли редагувати класи вузлів. У цьому випадку можна було б додати метод експорту до кожного типу вузлів, а потім, перебираючи всі вузли графа, викликати цей метод для кожного вузла. Завдяки поліморфізму, рішення було б елегантним, оскільки ви могли б не прив'язуватися до конкретних класів вузлів.

Але, на жаль, змінити класи вузлів у вас не вийшло. Системний архітектор сказав, що код класів вузлів зараз дуже ста-

більший, і від нього багато що залежить, а тому він не хоче ризикувати, дозволяючи будь-кому чіпати цей код.



*Код XML-експорту доведеться додати до всіх класів вузлів, а це дуже невигідно.*

До того ж він сумнівався в тому, що експорт до XML взагалі є доречним в рамках цих класів. Їхнє основне завдання пов'язане з геоданими, а експорт виглядає в межах цих класів, як біла ворона.

Була ще одна причина заборони. Наступного тижня вам міг знадобитися експорт в який-небудь інший формат даних, а це призвело б до повторних змін в класах.

## 😊 Рішення

Патерн Відвідувач пропонує розмістити нову поведінку в окремому класі, замість того, щоб множити її відразу в декількох класах. Об'єкти, з якими повинна бути пов'язана пове-

дінка, не виконуватимуть її самостійно. Замість цього ви будете передавати ці об'єкти до методів відвідувача.

Код поведінки, імовірно, повинен відрізнятися для об'єктів різних класів, тому й методів у відвідувача повинно бути декілька. Назви та принцип дії цих методів будуть подібними, а основна відмінність торкатиметься типу, що приймається в параметрах об'єкта, наприклад:

```

1 class ExportVisitor implements Visitor {
2     method doForCity(City c) { ... }
3     method doForIndustry(Industry f) { ... }
4     method doForSightSeeing(SightSeeing ss) { ... }
5     // ...

```

Тут виникає запитання, яким чином ми будемо подавати вузли до об'єкта відвідувача. Оскільки усі методи відрізняються сигнатурою, використати поліморфізм при перебираці вузлів не вийде. Доведеться перевіряти тип вузлів для того, щоб вибрати відповідний метод відвідувача.

```

1 foreach (Node node : graph)
2     if (node instanceof City)
3         exportVisitor.doForCity((City) node);
4     if (node instanceof Industry)
5         exportVisitor.doForIndustry((Industry) node);
6     // ...

```

Тут не допоможе навіть механізм перевантаження методів (доступний у Java і C#). Якщо назвати всі методи однаково, то невизначеність реального типу вузла все одно не дасть викликати правильний метод. Механізм перевантаження весь час викликатиме метод відвідувача, відповідний типу `Node`, а не реального класу поданого вузла.

Але патерн Відвідувач вирішує і цю проблему, використовуючи механізм подвійної диспетчеризації. Замість того, щоб самим шукати потрібний метод, ми можемо доручити це об'єктам, які передаємо в параметрах відвідувачеві, а вони вже самостійно викличуть правильний метод відвідувача.

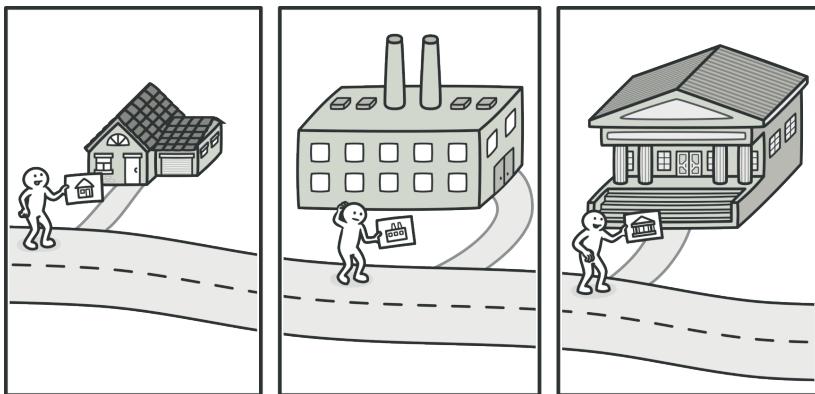
```

1 // Client code
2 foreach (Node node : graph)
3     node.accept(exportVisitor);
4
5 // City
6 class City is
7     method accept(Visitor v) is
8         v.doForCity(this);
9     // ...
10
11 // Industry
12 class Industry is
13     method accept(Visitor v) is
14         v.doForIndustry(this);
15     // ...

```

Як бачите, змінити класи вузлів все-таки доведеться. Проте ця проста зміна дозволить застосувати до об'єктів вузлів інші поведінки, адже класи вузлів будуть прив'язані не до конкретного класу відвідувачів, а до їхнього загально-го інтерфейсу. Тому, якщо доведеться додати до програми нову поведінку, ви створите новий клас відвідувачів і будете передавати його до методів вузлів.

## Аналогія з життя



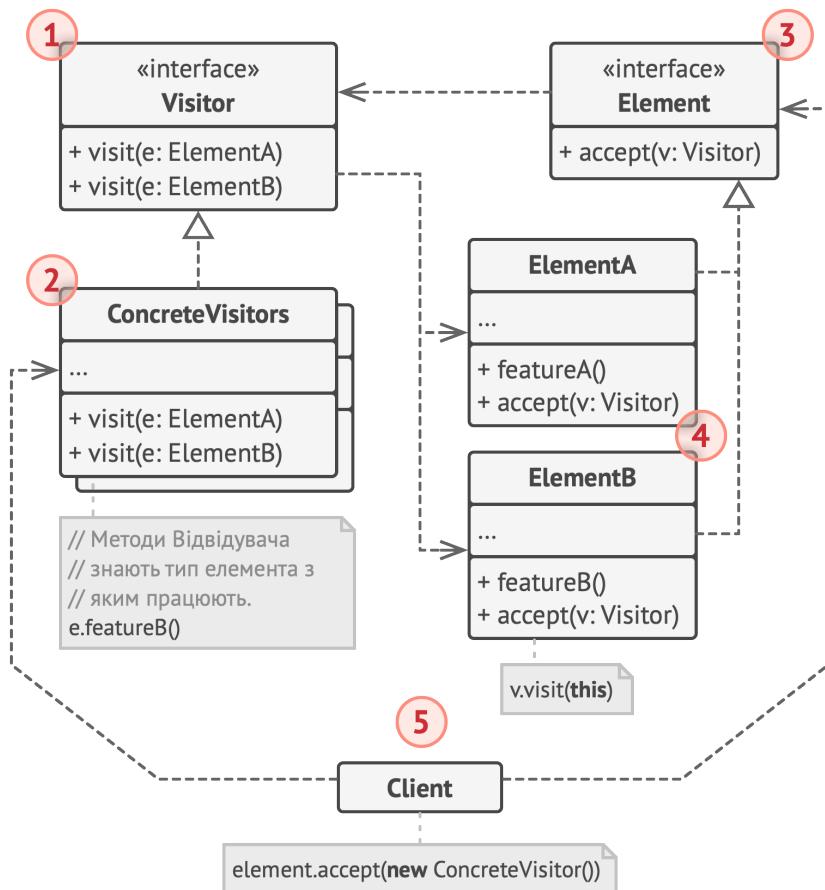
У страхового агента приготовані поліси для різних видів організацій.

Уявіть собі страхового агента-початківця, який прагне отримати нових клієнтів. Він хаотично відвідує всі будинки навколо, пропонуючи свої послуги. Але для кожного *типу* будинків, які він відвідує, у нього є особлива пропозиція.

- Прийшовши до будинку звичайної сім'ї, він пропонує оформити медичну страховку.

- Прийшовши до банку, він пропонує страховку на випадок пограбування.
- Прийшовши на фабрику, він пропонує страхування підприємства на випадок пожежі чи повені.

## Структура



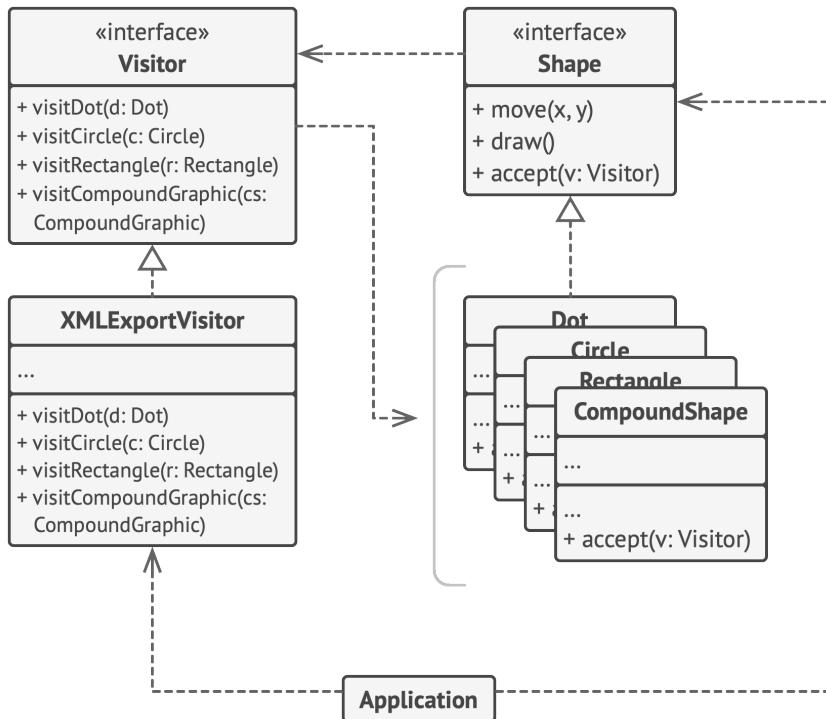
1. **Відвідувач** описує спільний для всіх типів відвідувачів інтерфейс. Він оголошує набір методів, що відрізняються типом

вхідного параметра. Кожному класу конкретних елементів повинен підходити свій метод. В мовах, які підтримують перевантаження методів, ці методи можуть мати однакові імена, але типи їхніх параметрів повинні відрізнятися.

2. **Конкретні відвідувачі** реалізують якусь особливу поведінку для всіх типів елементів, які можна подати через методи інтерфейсу відвідувача.
3. **Елемент** описує метод *прийому* відвідувача. Цей метод повинен мати лише один параметр, оголошений з типом загального інтерфейсу відвідувачів.
4. **Конкретні елементи** реалізують методи *приймання* відвідувача. Мета цього методу – викликати той метод відвідування, який відповідає типу цього елемента. Так відвідувач дізнається, з яким типом елементу він працює.
5. **Клієнтом** зазвичай виступає колекція або складний складовий об'єкт, наприклад, дерево **Компонувальника**. Здебільшого, клієнт не прив'язаний до конкретних класів елементів, працюючи з ними через загальний інтерфейс елементів.

## # Псевдокод

У цьому прикладі **Відвідувач** додає до існуючої ієархії класів геометричних фігур можливість експорту до XML.



Приклад організації експорту об'єктів XML через окремий клас-відвідувач.

```

1 // Складна ієрархія елементів.
2 interface Shape is
3     method move(x, y)
4     method draw()
5     method accept(v: Visitor)
6
7 // Метод прийняття відвідувача повинен бути реалізований у
8 // кожному елементі, а не тільки у базовому класі. Це допоможе
9 // програмі визначити, який метод відвідувача потрібно викликати
10 // у випадку, якщо ви не знаєте тип елемента.
11 class Dot implements Shape is
  
```

```

12  // ...
13  method accept(v: Visitor) is
14      v.visitDot(this)
15
16 class Circle implements Shape is
17  // ...
18  method accept(v: Visitor) is
19      v.visitCircle(this)
20
21 class Rectangle implements Shape is
22  // ...
23  method accept(v: Visitor) is
24      v.visitRectangle(this)
25
26 class CompoundShape implements Shape is
27  // ...
28  method accept(v: Visitor) is
29      v.visitCompoundShape(this)
30
31
32 // Інтерфейс відвідувачів повинен містити методи відвідування
33 // кожного елемента. Важливо, щоб ієархія елементів змінювалася
34 // рідко, оскільки при додаванні нового елемента доведеться
35 // змінювати всіх існуючих відвідувачів.
36 interface Visitor is
37     method visitDot(d: Dot)
38     method visitCircle(c: Circle)
39     method visitRectangle(r: Rectangle)
40     method visitCompoundShape(cs: CompoundShape)
41
42 // Конкретний відвідувач реалізує одну операцію для всієї
43 // ієархії елементів. Нова операція = новий відвідувач.

```

```
44 // Відвідувача вигідно застосовувати, коли нові елементи
45 // додаються дуже зрідка, а нові операції – часто.
46 class XMLExportVisitor implements Visitor is
47     method visitDot(d: Dot) is
48         // Експорт id та координат центру точки.
49
50     method visitCircle(c: Circle) is
51         // Експорт id, координат центру та радіусу кола.
52
53     method visitRectangle(r: Rectangle) is
54         // Експорт id, координат лівого-верхнього кута, висоти
55         // та ширини прямокутника.
56
57     method visitCompoundShape(cs: CompoundShape) is
58         // Експорт id складової фігури, а також списку id
59         // підфігур, з яких вона складається.
60
61
62 // Програма може застосовувати відвідувача до будь-якого набору
63 // об'єктів елементів, навіть не уточнюючи їхні типи. Потрібний
64 // метод відвідувача буде обрано завдяки проходу через метод
65 // accept.
66 class Application is
67     field allShapes: array of Shapes
68
69     method export() is
70         exportVisitor = new XMLExportVisitor()
71
72         foreach (shape in allShapes) do
73             shape.accept(exportVisitor)
```

Вам не здається, що виклик методу `accept` – це зайва ланка? Якщо так, тоді ще раз рекомендую вам ознайомитися з проблемою раннього та пізнього зв'язування в статті [Відвідувач і Double Dispatch](#).

## Застосування

-  Якщо вам потрібно виконати якусь операцію над усіма елементами складної структури об'єктів, наприклад, деревом.
-  Відвідувач дозволяє застосовувати одну і ту саму операцію до об'єктів різних класів.
-  Якщо над об'єктами складної структури об'єктів потрібно виконувати деякі не пов'язані між собою операції, але ви не хочете «засмічувати» класи такими операціями.
-  Відвідувач дозволяє витягти споріднені операції з класів, що складають структуру об'єктів, помістивши їх до одного класу-відвідувача. Якщо структура об'єктів використовується в декількох програмах, то патерн дозволить кожній програмі мати тільки потрібні в ній операції.
-  Якщо нова поведінка має сенс тільки для деяких класів з існуючої ієархії.
-  Відвідувач дозволяє визначити поведінку тільки для цих класів, залишивши її порожньою для всіх інших.

## Кроки реалізації

1. Створіть інтерфейс відвідувача й оголосіть у ньому методи «відвідування» для кожного класу елемента, який існує в програмі.
2. Опишіть інтерфейс елементів. Якщо ви працюєте з уже існуючими класами, оголосіть абстрактний метод прийняття відвідувачів у базовому класі ієрархії елементів.
3. Реалізуйте методи прийняття в усіх конкретних елементах. Вони повинні переадресовувати виклики тому методу відвідувача, в якому тип параметра збігається з поточним класом елемента.
4. Ієрархія елементів повинна знати тільки про загальний інтерфейс відвідувачів. З іншого боку, відвідувачі знатимуть про всі класи елементів.
5. Дляожної нової поведінки створіть свій власний конкретний клас. Пристосуйте цю поведінку для роботи з усіма наявними типами елементів, реалізувавши всі методи інтерфейсу відвідувачів.

Ви можете зіткнутися з ситуацією, коли відвідувачу потрібен доступ до приватних полів елементів. У цьому випадку ви можете або розкрити доступ до цих полів, порушивши інкапсуляцію елементів, або зробити клас відвідувача вкладеним в клас елемента, якщо вам пощастило писати мовою, яка підтримує механізм вкладених класів.

- Клієнт створюватиме об'єкти відвідувачів, а потім передаватиме їх елементам через метод прийняття.

## Переваги та недоліки

- ✓ Спрошує додавання операцій, працюючих зі складними структурами об'єктів.
- ✓ Об'єднує споріднені операції в одному класі.
- ✓ Відвідувач може накопичувати стан при обході структури елементів.
- ✗ Патерн невіправданий, якщо ієрархія елементів часто змінюється.
- ✗ Може привести до порушення інкапсуляції елементів.

## Відносини з іншими патернами

- Відвідувач можна розглядати як розширений аналог Команди, що здатен працювати відразу з декількома видами одержувачів.
- Ви можете виконати якусь дію над усім деревом Компонувальника за допомогою Відвідувача.
- Відвідувач можна використовувати спільно з Ітератором. Ітератор відповідатиме за обхід структури даних, а Відвідувач — за виконання дій над кожним її компонентом.

# Заключення

## Вітаю! Ви дісталися закінчення!

Але у світі існує безліч інших патернів. Сподіваюся, ця книга стане вашою точкою старту в подальшому оволодінні патернами та розвитку надзвичайних здібностей у проектуванні програм.

Ось декілька ідей для наступних кроків, якщо ви ще не визначилися з тим, що робитимете далі:

- Не забувайте, що разом з цією книгою поставляється архів з реальними прикладами коду різними мовами програмування.
- Прочитайте книгу Джошуа Керієвські [Рефакторинг з використанням патернів проектування](#).
- Не розбираєтесь у рефакторингу? [У мене є хороші матеріали для вас на Refactoring.Guru](#).
- Роздрукуйте [шпаргалки по патернах](#) та прикріпіть їх десь на видному місці.
- [Залиште відгук](#) про цю книгу. Мені було б дуже цікаво почути вашу думку, навіть якщо це критика 😊